

Práctica 3 “Algoritmos Voraces (Greedy)”

Parte II: El problema del viajante de comercio



Realizado por:

Baeza Álvarez, Jesús
García Moreno, Jorge
López Maldonado, David
Rodríguez Calvo, Jose Manuel
Sánchez Molina, Alejandro

ÍNDICE

- 1. Objetivo del problema**
- 2. Explicación algoritmo 1: Vecino más cercano.**
- 3. Explicación algoritmo 2: Inserción.**
- 4. Explicación algoritmo 3: Propuesta del grupo, 2-opt**
- 5. Estudio comparativo de algoritmos.**
- 6. Bibliografía**

1. Objetivo del problema

El objetivo de esta práctica era implementar 3 programas que proporcione soluciones para el problema del viajante de comercio empleando las heurística del vecino más cercano, inserción, así como otra adicional propuesta por el propio equipo.

Además también se tiene por objetivo el realizar un estudio comparativo de las tres estrategias empleando un conjunto de datos de prueba.

Para la visualización de los datos se utilizó gnuplot. Los datos utilizados han sido proporcionados por el profesor en la plataforma decsai, así como un programa para la reordenación de los puntos obtenidos.

Para ello, definimos lo siguiente:

VECINO MÁS CERCANO

Conjunto de candidatos (C): Diferentes ciudades a visitar

-Conjunto de seleccionados (S): Ciudades ya visitadas

-Función solución: Esta función tiene que verificar que se han visitado todas las ciudades.

-Función selección: Aquella función que devuelve la ciudad con menor distancia a la ciudad actual del conjunto de ciudades aún no visitadas.

-Función objetivo: Minimizar la distancia total del recorrido.

INSERCIÓN

Conjunto de candidatos (C): Diferentes ciudades a visitar

-Conjunto de seleccionados (S): Ciudades ya visitadas

-Función solución: Esta función tiene que verificar que se han visitado todas las ciudades.

-Función selección: Aquella función que devuelve la ciudad, la cual si la añadimos al ciclo de una distancia menor.

-Función objetivo: Minimizar la distancia total del recorrido.

2. Explicación algoritmo 1: Vecino más cercano.

Partiendo de los datos proporcionados en la práctica, es decir, de los ficheros con el número de cada ciudad y con sus coordenadas X e Y correspondientes.

Comenzamos en la función leer_puntos, en la cual leemos el fichero de texto y utilizando la fórmula proporcionada en el guión, convertimos los datos de las ciudades a una matriz, en la cual introducimos la distancia correspondiente a cada ciudad con respecto a las demás ciudades.

Para calcular el recorrido más corto, hemos implementado el método Greedy, en el cual realizamos los siguientes pasos:

Primero: Rellenamos de 0 la columna correspondiente a la ciudad de la que partimos, teniendo en cuenta que cada fila y columna de la matriz corresponde a una ciudad y que cada elemento de la matriz corresponde a la distancia que hay entre una ciudad y otra (la fila 0 y columna 0 de la matriz corresponde a la distancia que hay de la ciudad 0 hasta la ciudad 0 en este caso 0 por eso la diagonal principal siempre es 0), poniendo el valor de todos los datos de sus columnas a 0 indicamos que no podemos repetir dicha ciudad, teniendo en cuenta que trabajamos con una copia de la matriz de las ciudades y que en el cálculo de las distancias no tomamos el 0 como una distancia viable.

Segundo: En base a la matriz con las distancias correspondientes a cada una de las ciudades, tomamos cada fila de ésta como un vector, y calculamos el mínimo valor de éste en la función MinimoVector sin tener en cuenta los 0 . Una vez obtenido el mínimo, nos vamos a la fila a la que corresponde dicho valor, además de la distancia, tomamos también el número de la columna a la que corresponde, teniendo en cuenta que si se trata de la columna 5, quiere decir que la ciudad más cercana sería la ciudad 5 y que, por lo tanto debemos partir de la fila 5 (la siguiente vez que llamemos a greedy por recurrencia comenzará a buscar el mínimo de la fila 5). Una vez calculado la distancia mínima, añade dicha distancia a una variable que corresponde con la suma de la distancia total de todas las ciudades que hemos ido tomando, por último añadimos a un vector de int las ciudades que vamos recorriendo.

Tercero: En base al método Greedy, obtenemos un vector con el número de las ciudades que corresponderán a la distancia total comenzando desde la primera ciudad, lo añadimos a un vector, y volvemos a ejecutar el método Greedy pero partiendo desde la segunda ciudad, y así hasta haber recorrido todas las ciudades. Esto lo hacemos dentro de un bucle for para que empiece por la ciudad 1 y haga el método greedy comenzando por todas las ciudades.

Por último en el main fuera de la función Greedy calculamos el mínimo de este vector último (que contiene la distancia total comenzando desde la ciudad 1 , desde la ciudad 2 , ...) de forma que sabemos la ciudad por la que comenzar a la hora de ejecutar Greedy ya que sabemos que es la que su distancia total es menor.

Por lo que una vez más volvemos a ejecutar Greedy pero comenzando desde la ciudad obtenida en el método anterior y esta vez sí que el vector ciudades que ha ido guardando el recorrido correcto.

Para concluir imprimimos la distancia total de este recorrido y las ciudades por las que vamos visitando.

3. Explicación algoritmo 2: Inserción.

Para resolver el segundo apartado se ha usado el algoritmo propuesto de inserción. Previamente al algoritmo se leen los puntos y se crea la matriz de distancia tal como se ha explicado en el algoritmo anterior.

Primeramente en el vector resultado hemos introducido 3 ciudades formando un ciclo, el criterio ha sido coger la ciudad más al norte, la más al este y la más al oeste.

Después mientras haya ciudades sin recorrer(`while resultado.size() != NumeroCiudades`) se hace lo siguiente:

- `SeleccionarCiudad` para introducir al ciclo, en esta función se recorren todas las ciudades no visitadas, y para cada una se inserta en el ciclo se mide la distancia total del ciclo que genera y se saca, por último la ciudad cuya distancia generada haya sido menor es la que el algoritmo selecciona como siguiente ciudad a insertar.
- `MejorPos`, es una función que coge la ciudad seleccionada y mide la distancia del recorrido varias veces, una en cada posición posible en el recorrido que ya tenemos y la función devuelve la posición de la ciudad seleccionada en la cual de menor distancia.
- `InsertarCiudad` coge la ciudad seleccionada y la inserta en la mejor posición dada por la función anterior, generando así un ciclo con una ciudad más.

4. Explicación algoritmo 3: Propuesta del grupo.

“Heurística 2-opt”

Para resolver el tercer apartado de la práctica, el cual se nos pedía que utilizáramos un algoritmo libre para resolver el tsp, hemos empleado la heurística 2-opt, la cual es una técnica de optimización de una solución ya dada.

Explicación de cómo funciona este método:

Este algoritmo, en esencia coge una solución al problema del viajante y la mejora. En nuestro caso hemos cogido la solución dada por el método del vecino más cercano y le hemos aplicado el algoritmo 2-opt.

A ese vector resultado (llamado V) se le aplica la función DosOpt, en la cual sobre ese vector se sitúan dos punteros, i y j, i toma el valor desde la primera ciudad hasta penúltima y para cada valor de i, j toma valores desde i+1 hasta la última ciudad. Ahora se altera el vector V para conseguir otro recorrido de la siguiente manera:

- desde la ciudad inicial hasta V[i] se inserta normalmente en el nuevo recorrido.
- desde V[i] hasta V[j] esas ciudades son añadidas al nuevo recorrido al revés.
- y desde V[j] hasta la última ciudad se añade normalmente.

A este nuevo recorrido se le mide la distancia total y si es menor que la solución encontrada anteriormente, se actualiza la nueva ruta, puesto que posee mejor distancia.

Todo el proceso descrito anteriormente se repite hasta que el algoritmo no encuentra una mejor ruta que la actual, en nuestro caso 20 repeticiones sin mejora implica acabar el algoritmo.

5. Estudio comparativo de algoritmos.

En este apartado se va a presentar un estudio comparativo para los resultados obtenidos para cada uno de los algoritmos.

La forma de comparación será mediante una tabla en la que hemos recogido los resultados para cada uno de los algoritmo y el resultado óptimo.

Además también mostramos una comparación entre las gráficas obtenidas con gnuplot sobre uno de los archivos, observando así de forma visual, cuál de los algoritmos se ha acercado de forma más cercana al resultado óptimo.

Hemos probado sobre 10 ficheros para asegurarnos de que los algoritmos funcionan correctamente y podemos fiarnos de los datos obtenidos.

A continuación se muestra una tabla en la cual se plasma una comparación entre las distancia obtenidas para cada uno de los algoritmos.

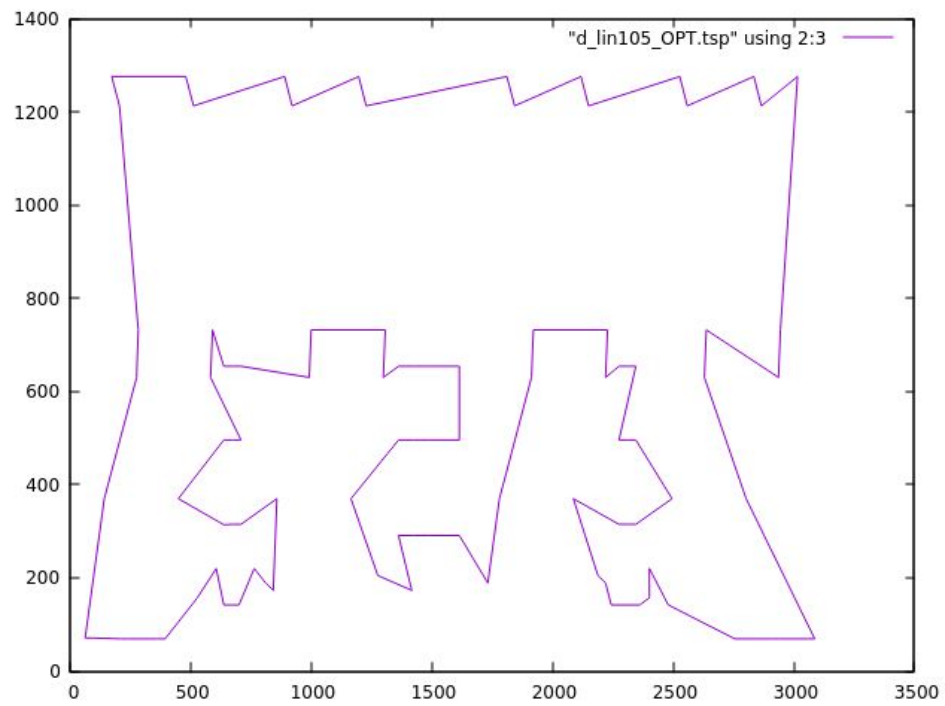
Fichero	Versión óptima	Vecino más cercano	Inserción	2-opt
a280.tsp	2579	3095.17	3049.28	2788.99
ch130.tsp	6110	7198.74	6555.34	6416.98
kroC100.tsp	20749	23566.4	23401.6	21939.5
eil51.tsp	426	505.77	490.75	449.33
lin105.tsp	14379	16959	16285.1	15072.2
tsp225.tsp	3916	4633.22	4499.19	4182.79
eil101.tsp	629	736.36	687.071	690.36
pcb442.tsp	50778	58953	57970.7	54283.3
berlin52.tsp	7542	8182.19	8286.87	7864.44
rd100.tsp	7910	9427.33	9212.22	8655.46

Como podemos observar, la heurística 2-opt, mejora la distancia en comparación con los otros algoritmos implementados y se acerca bastante a la distancia óptima.

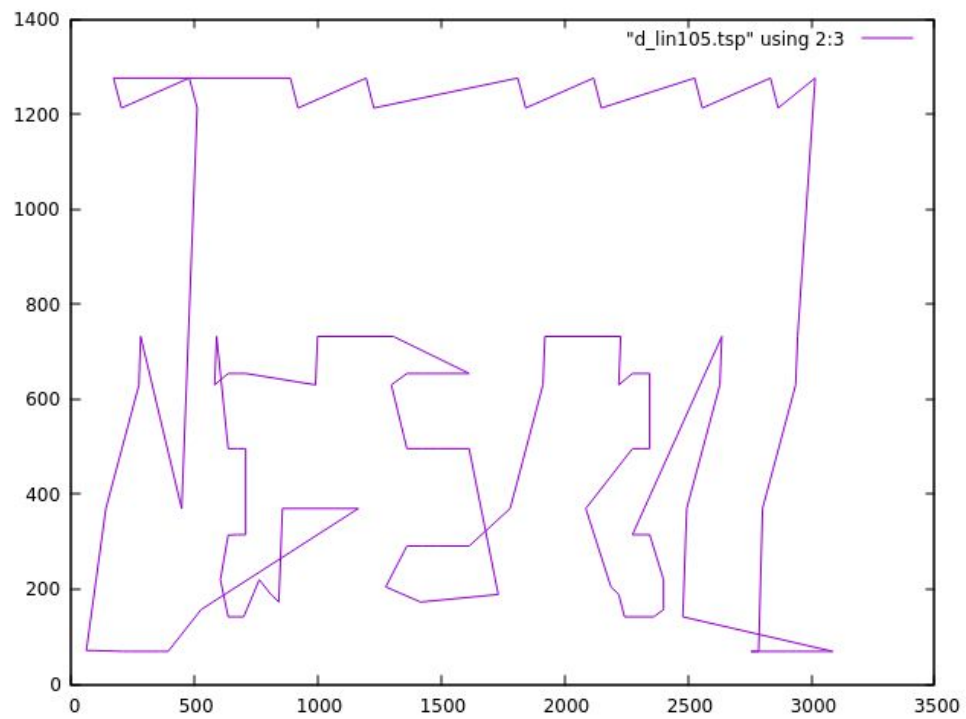
Además en la mayoría de los casos el algoritmo de inserción da mejor resultado que el vecino más cercano

Esta comparación se puede ver de forma más directa usando las gráficas obtenidas con gnuplot que a continuación se muestra. Usamos como ejemplo las gráficas del recorrido obtenidas para el fichero "lin105.tsp"

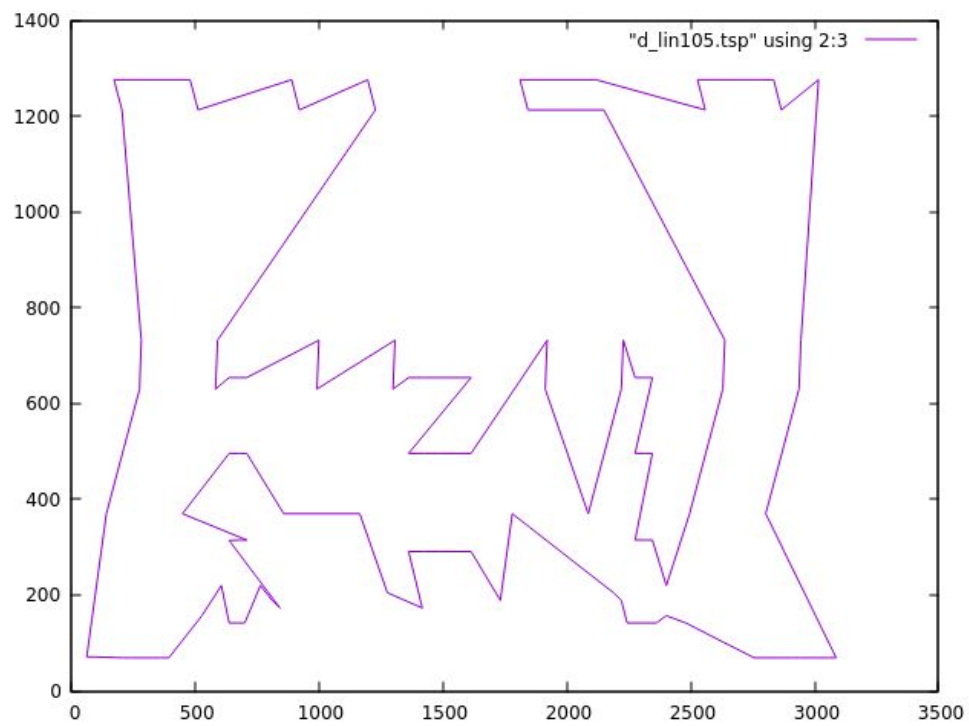
Versión óptima (según los datos proporcionados en el .html de la práctica)



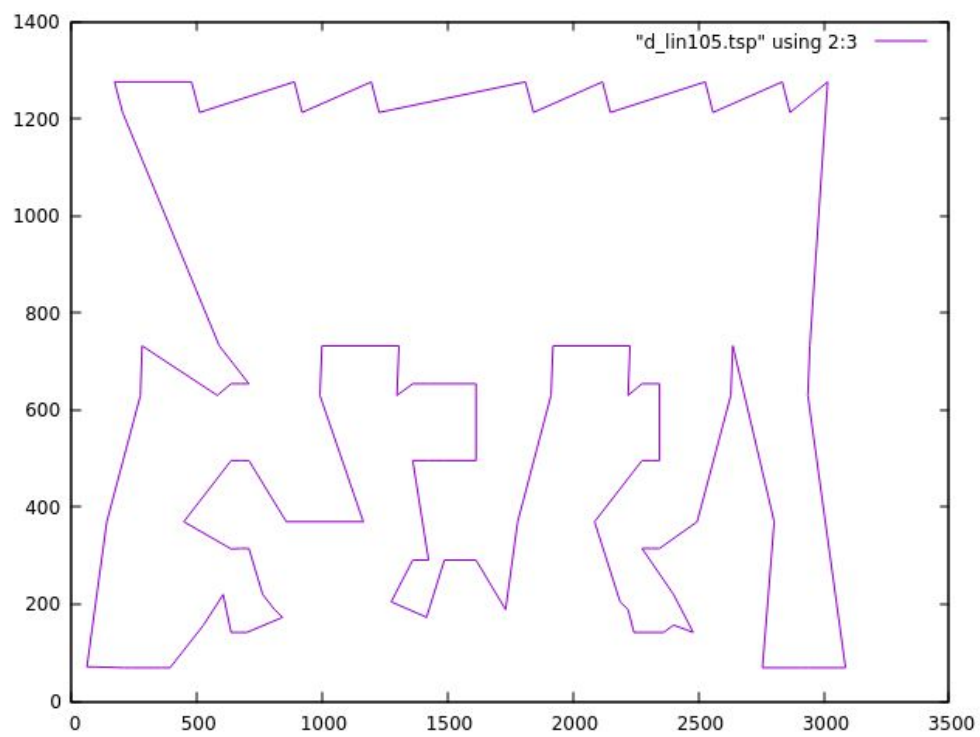
Versión vecino más cercano



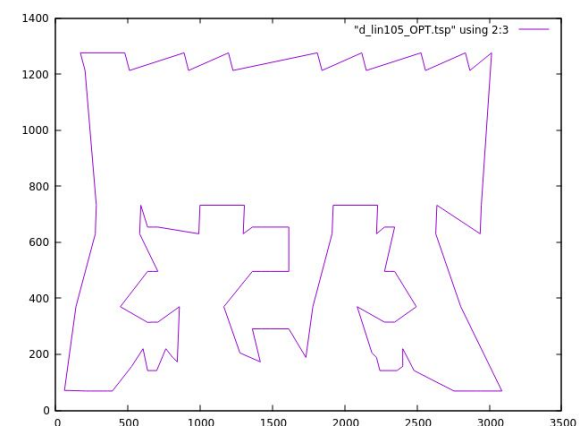
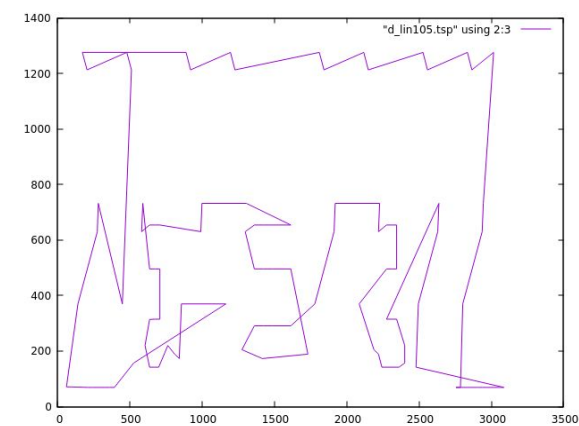
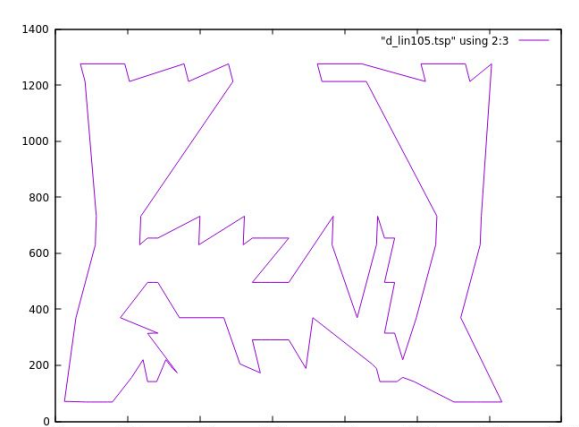
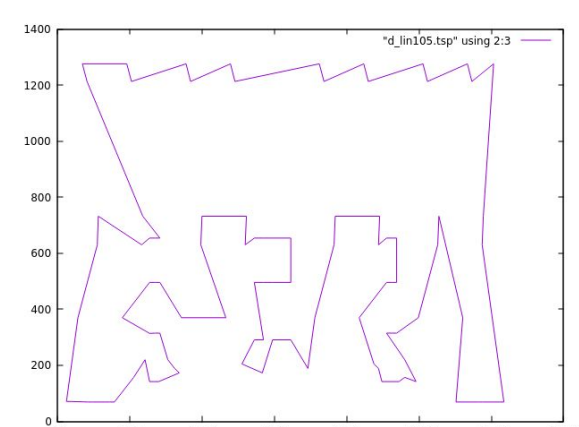
Versión inserción



Versión 2-opt



En esta tabla se puede ver de forma más clara la diferencia entre las gráficas de los recorridos.

 <p>A line graph showing an optimal route. The x-axis ranges from 0 to 3500, and the y-axis ranges from 0 to 1400. The route is a purple line that starts at (0,0), goes to (3000,0), then follows a complex path through several points, ending at (3000,1250). The path is relatively smooth and follows the outer boundary of the points.</p>	 <p>A line graph showing a nearest neighbor route. The x-axis ranges from 0 to 3500, and the y-axis ranges from 0 to 1400. The route is a purple line that starts at (0,0), goes to (3000,0), then follows a complex path through several points, ending at (3000,1250). The path is more jagged and less smooth than the optimal route.</p>
Versión óptima (según los datos proporcionados en el .html de la práctica)	Versión vecino más cercano
 <p>A line graph showing an insertion route. The x-axis ranges from 0 to 3500, and the y-axis ranges from 0 to 1400. The route is a purple line that starts at (0,0), goes to (3000,0), then follows a complex path through several points, ending at (3000,1250). The path is more jagged and less smooth than the optimal route.</p>	 <p>A line graph showing a 2-opt route. The x-axis ranges from 0 to 3500, and the y-axis ranges from 0 to 1400. The route is a purple line that starts at (0,0), goes to (3000,0), then follows a complex path through several points, ending at (3000,1250). The path is more jagged and less smooth than the optimal route.</p>
Versión inserción	Versión 2-opt

6. Bibliografía

https://es.wikipedia.org/wiki/Problema_del_viajante

<http://materias.fi.uba.ar/7114/Docs/ApunteHeurísticas.pdf>

Apuntes de clase y transparencias proporcionadas por el profesor.