UNIVERSIDAD DE GRANADA
E.T.S. DE INGENIERÍAS INFORMÁTICA Y DE TELECOMUNICACIÓN

Asignatura: Algorítmica

Grupo nº: 4

Integrantes: Raúl Rodríguez Pérez, Francisco Javier Gallardo Molina, Inés Nieto Sánchez, Antonio Lorenzo Gavilán Chacón

Grupo: C1

Resolución Práctica 4: Programación Dinámica

Curso 2019-2020

Grado en Ingeniería Informática

Índice:

1. Introducción…… pág 3
2. Enfoque basado en PD…….. …… pág 3
3. Descripción del algoritmo …..…… pág 3
    a. Descripción de la función g ……..…….. pág 4
    b. Pseudocódigo del problema ……... pág 5
    c. Descripción de la recomposición ……..……. pág 5
4. Ejemplos de ejecución ……...…… pág 6
    a. Ulysses16 …….. pág 6
    b. Ulysses22 …….. pág 8
5. Comparación PD-Greedy …..……..... pág 10

1. Introducción

En esta cuarta práctica el tema a abordar es la programación dinámica. Este método se basa en la utilización de la suma de versiones más pequeñas del problema original o subproblemas, cuando se emplea de forma recursiva. Como podemos apreciar, este método se asemeja bastante a los algoritmos de Divide y Vencerás, puesto que también se basa en la descomposición del problema en subproblemas y combinar esas soluciones parciales en una solución global. Aunque estos dos tipos de algoritmos sean muy similares, tienen una clara diferencia, la programación dinámica descompone el problema en distintos subproblemas, los cuales solo se resuelven una vez y se guardan en una tabla o matriz, por lo que si es necesario obtener esos datos de nuevo solo hay que consultarlos. Siendo la solución al problema inicial es una combinación de estos datos almacenados. Dicho procedimiento que se basa en almacenar los datos en una tabla o matriz se conoce como memoria o memorización.

En esta práctica se nos pide resolver de nuevo el problema del viajante de comercio, pero esta vez, utilizando programación dinámica. Tras completar la implementación del código, se no pide comparar los resultados obtenidos con los de la práctica anterior con el enfoque Greedy o voraz.

2. Enfoque basado en programación dinámica

Para explicar cómo hemos desarrollado nuestro código para resolver el problema del viajante de comercio, primero vamos a hablar brevemente de la siguiente cuestión.

Uno de los problemas que poseen los algoritmos recursivos, se basa en que durante la ejecución se suelen repetir los mismos cálculos en más de una ocasión. Para ejemplificar esto visualicemos el algoritmo de Fibonacci, dicho algoritmo se puede implementar de manera recursiva definiendo la función como f(x)=f(x-1)+f(x-2) si x>1. Sí nosotros quisiéramos calcular f(4), debemos calcular previamente f(2) y f(3), y a su vez para obtener f(3), debemos calcular f(2)-f(1)-f(0) y f(1)-f(0). Como se puede ver, cálculos como f(2) o f(1) se repiten en más de una ocasión. Tal y como hemos introducido antes, la programación dinámica consigue paliar este problema utilizando la memorización. Es decir, calculamos f(2) y f(1) una única vez, así cuando haga falta su cálculo en posteriores ejecuciones, simplemente se utilizará el resultado almacenado en memoria. Este caso es el que aprovechamos para implementar nuestro algoritmo, utilizamos una función recursiva ‘g’, que almacena los resultados en una matriz, aprovechándose así de la programación dinámica.

Cabe destacar que, la solución más óptima de implementar el algoritmo de fibonacci es mediante un algoritmo iterativo en la que f(x), sea la suma de f(x-1) + f(x-2), pero simplemente queríamos ilustrar el enfoque que hemos elegido a la hora de realizar nuestro algoritmo.

3. Descripción del algoritmo

Para comprender el funcionamiento de este algoritmo es necesario previamente entender la forma en la que se representan los distintos subconjuntos que se pueden formar con el conjunto de todas las ciudades.

Para explicarlo mejor supongamos el ejemplo de un conjunto de 4 ciudades {1,2,3,4} al que se le aplicará el algoritmo para calcular el recorrido que da la distancia mínima al visitarlas de una en una.

El número de subconjuntos que se pueden formar con estas ciudades es igual a 2^(4-1) = 2^3 = 8 subconjuntos. Puede darse el caso de que existan más de un subconjunto formado por dos elementos y si esto ocurre ¿cómo los distinguimos?.

La respuesta es muy sencilla, se utiliza una notación de bits de un entero que representa al subconjunto. Por ejemplo, los subconjuntos que se pueden formar con las cuatro ciudades descritas anteriormente son los siguientes ({ }, {2}, {3}, {4}, {2,3}, {2,4}, {3,4}, {2,3,4}). Para referirnos a ellos se utilizará el siguiente código binario:

{ }	=	000
{2}	=	001
{3}	=	010
{4}	=	100
{2,3}	=	011
{2,4}	=	101
{3,4}	=	110
{2,3,4}	=	111

El subconjunto vacío se representa con todos los bits a cero y el subconjunto {2,3,4} se anota con todos los bits a 1 indicando que es el mayor de los subconjuntos. Para los demás subconjuntos se rellenan los bits siguiendo un orden definido.

De este modo las operaciones que se hagan en el algoritmo como el AND lógico o el desplazamiento de bits (<<) se harán en función de estos valores que se hacen a los subconjuntos de las ciudades.

Para la explicación del algoritmo que hemos implementado, vamos a dividir la explicación según las funciones más importantes que hemos creado, es decir, explicaremos con detalle tanto la función ‘g’, la cual calcula la distancia mínima o la distancia del recorrido óptimo, como la reconstrucción de la solución a raíz de la matriz que se rellena en la función g, es decir, el recorrido óptimo con sus ciudades.

3.1 Descripción de la función g

Para la resolución de nuestro problema hemos implementado una función llamada ‘g’, la cual es una función recursiva (utilizando programación dinámica tal y como hemos explicado anteriormente) que calcula la longitud del camino mínimo

partiendo desde un vértice llamado “pos” pasando por todas las ciudades del conjunto “visitadas” volviendo de nuevo al vértice 1.

Dicha función se apoya en una matriz que llamaremos matriz solución (o gtab en el pseudocódigo), la cual se irá rellenando con todos los valores de los mínimos que irá calculando la función. Los parámetros de ‘g’ son: la matriz de distancias, la posición en la que nos encontramos, el conjunto de ciudades y la matriz solución.

Procederemos ahora a realizar una explicación más exhaustiva de la función por medio del pseudocódigo:

```
function g(i, S)
  if S = ∅ then return L[i, 1]
  if gtab[i, S] ≥ 0 then return gtab[i, S]
  ans ← ∞
  for each j ∈ S do
    distviaj ← L[i, j] + g(j, S \ {j})
    if distviaj < ans then ans ← distviaj
  gtab[i, S] ← ans
  return ans ,
```

Aclaraciones: L -> matriz de distancias, gtab -> matriz solución

El pseudocódigo se basa en la utilización de la función g(i, S) que devuelve la mínima distancia que hay entre la ciudad “i” y la ciudad “1” pasando por todas las ciudades que están incluidas en el conjunto “S”.

En primer lugar, se toma como base el valor de la función g(i, 0), que se corresponde con la distancia que existe entre la ciudad “i” y la ciudad “1” sin pasar por ninguna otra ciudad (en este caso el conjunto S está vacío). Esta distancia es igual a la que nos proporciona la matriz de distancias L[i, 1], de la que obtenemos el valor de la distancia como el valor de la fila “i” y la columna “0” de dicha matriz.

Se utiliza la matriz gtab[i, S] que es una matriz de orden “n x 2^(n-1)” donde n es el número de ciudades y 2^(n-1) es el número de subconjuntos de ciudades que se pueden hacer con el conjunto S formado por “n” ciudades. Esta matriz tiene en principio todos sus valores igual a -1.

Si resulta que el valor de la matriz gtab[i, S] para la ciudad “i” es mayor o igual que cero entonces se devuelve este valor, esto es así ya que, al inicializar la matriz a -1, que el valor en dicha posición sea mayor que 0 significa que se ha modificado, por lo cual, ya está puesta la distancia mínima que buscábamos para [i,S].

Si no es así, para cada ciudad “j” perteneciente al conjunto S se calcula recursivamente la suma de la distancia que hay desde la ciudad “i” a la “j” y la distancia mínima desde la ciudad “j” hasta la ciudad “1” pasando por todas las ciudades del conjunto S original habiendo eliminado previamente la ciudad “j” de este conjunto. Si esta suma es menor que el valor mínimo (“ans”) que inicialmente estaba a infinito entonces se actualiza el valor de esta matriz con esta distancia y al final se devuelve el mínimo de todas las distancias que empezando por la ciudad “i” recorren todas las ciudades del conjunto S y terminan en la ciudad “1”, que concuerda con el valor de gtab [i, S].

3.2 Descripción de la recomposición del recorrido

La otra función importante que queremos explicar más profundamente es aquella que reconstruye (a raíz de la matriz solución) el recorrido óptimo del problema. Los parámetros que incluye esta función son la matriz solución “m” de tamaño “n x 2^(n-1)” donde “n” es el número de ciudades y que tiene los valores de las mínimas distancias calculadas con la función “g”; el segundo parámetro es el entero “todas_visitadas” que se corresponde con el número de ciudades que quedan por visitar para formar el recorrido; y como tercer parámetro tenemos la matriz de distancias “L”, que nos da las distancias entre cada par de ciudades.

En general para tratar de sintetizar que hace la función, esta se basa en, tras haber conseguido rellenar la matriz solución y obtener la distancia del recorrido óptima vamos a volver sobre nuestros pasos para conseguir averiguar cuales son las ciudades que componen dicho recorrido y su orden. Lo de volver sobre nuestros pasos hacer referencia a que, partiendo de g(i,S), es decir, desde la última ejecución de la función g, vamos a dar marcha atrás con ayuda de la matriz solución. Por ejemplo, sí partimos de g(4,{2,3}), sabemos que esto es igual a encontrar el min(L42 + g(2,{3}), L43 + g(3,{2})). Lo que quiere hacer nuestro algoritmo es ir dando los pasos al contrario que la función recursiva g, es decir, va analizando cuáles han sido los valores que se han convertido en min para los diferentes conjuntos de ciudades, por que una vez los tenga, se podrá conocer la ciudad a la cual se ha desplazado, y la ciudad desde la que se ha desplazado.

Haciendo una mayor énfasis en el código, la función se ejecuta mientras que existan ciudades disponibles para visitar, estas las recorre desde la ciudad 1 hasta la última, y siempre que la ciudad recorrida no haya sido visitada anteriormente y no haya sido escogida en una búsqueda anterior como parte de la solución.

Entonces se comprueba si el valor de la matriz “m” para dicha ciudad es menor que el mínimo valor prefijado inicialmente a infinito. Si esto ocurre entonces se actualiza el valor del mínimo con ese valor y se guarda la posición en la que ocurre ese mínimo, que se corresponde con el número de la ciudad recorrida hasta el momento. Dicha ciudad se añade a la lista de ciudades que es la solución y que

nos da el recorrido final, repitiendo el ciclo hasta que ya no haya ciudades disponibles que visitar.

Queremos destacar que para comprobar que nuestra función realizaba su trabajo correctamente, hemos buscado diversos ejemplos por internet donde nos daban una matriz de distancias y nos ponían el recorrido que debería salir empleando PD. Por ello en el código aparecen una series de matrices que hacen referencia a esta página web:

https://www.slideshare.net/luisalfredomoctezumapascual/el-problema-del-agente-viajero-resuelto-por-fuerza-programacin-dinmica-y-voraz

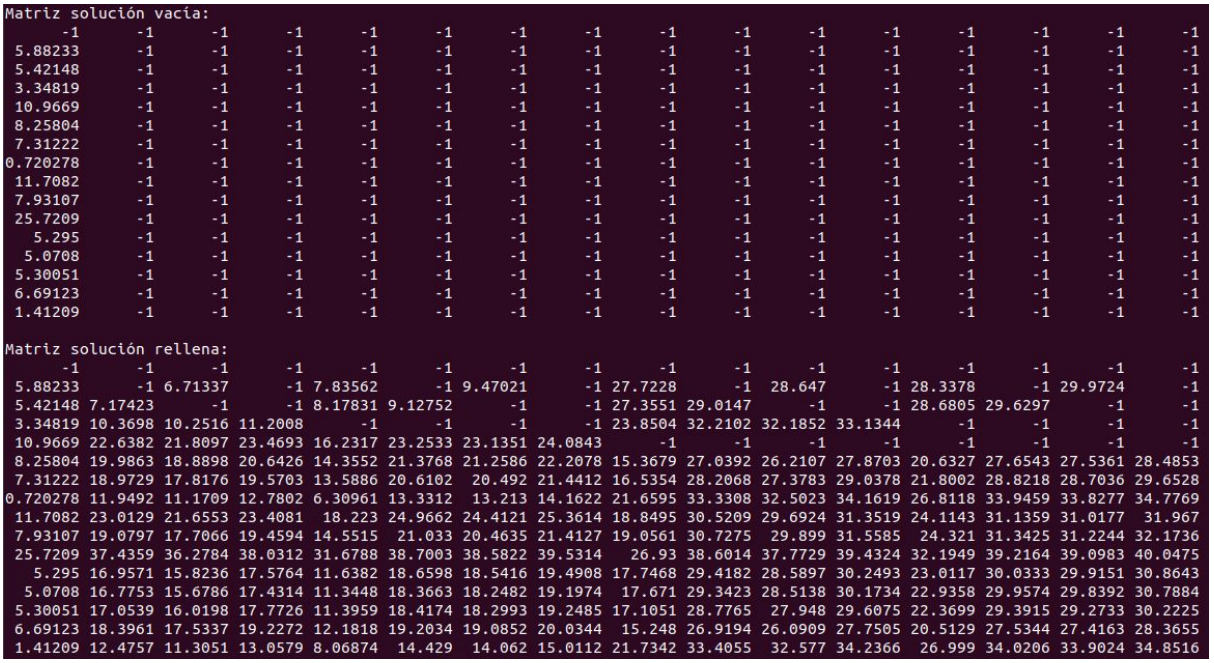
En ella aparece los ejemplos que acabamos de comentar, para una matriz de 5*5, 8*8 y 10*10. Cabe destacar que para la matriz de 5*5 no da exactamente el recorrido que sale en la página, pero sí compruebas a mano el recorrido que nos aparece cumple la distancia mínima, por lo que también es un recorrido óptimo. Queremos recalcar que esta no es la única página con la que hemos comprobado esto, pero sí nos pareció la que mejor y más fácil nos dejaba las cosas.

4. Ejemplos de ejecución

- **ULYSSES 16**

Hemos realizado una ejecución del algoritmo con los datos proporcionados en el archivo “ulysses16”, en el cual, nos vienen 16 ciudades con sus respectivas coordenadas. Vamos a proceder a plasmar los resultados tras haber explicado su funcionamiento.

Como ya hemos comentado, en primer lugar vamos a inicializar la matriz solución a -1, y llamaremos a la función ‘g’, que rellenandola, conseguirá obtener la distancia del recorrido óptimo. Vamos a mostrar como queda la matriz, pero cabe destacar que al tratarse de una matriz de tamaño n*2^(n-1), no se puede mostrar en su plenitud. Aun así hemos querido mostrar la matriz hasta el máximo que se nos permite:



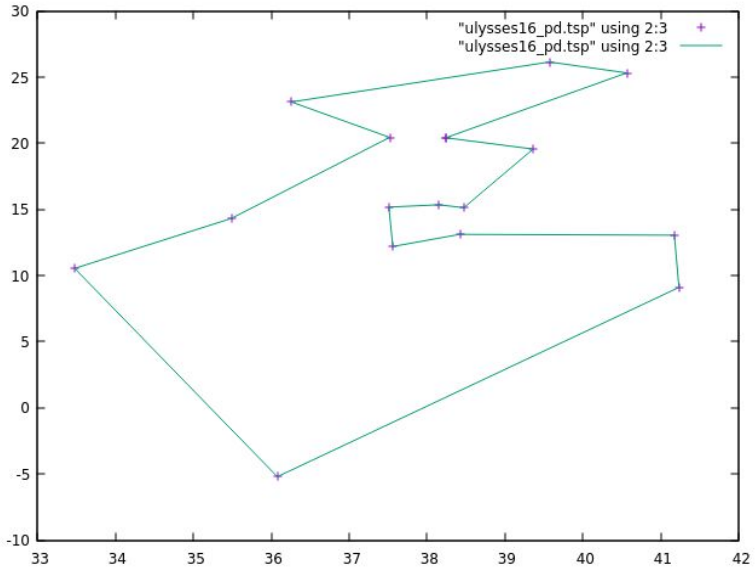
Aprovechando la imagen vamos a explicar algo que no hemos dicho con anterioridad, en la primera matriz se ve inicializada la primera columna, cosa que no tendría sentido porque hemos dicho que se inicializa enteramente a -1. Esto se debe a que, la primera columna de la matriz solución hace referencia al conjunto vacío por lo que, la distancia que hay desde una ciudad al conjunto vacío, sería la distancia que hay desde dicha ciudad a la primera. Y esto corresponde con la primera columna de la matriz de distancia, por ello en el codigo despues de inicializar a -1 la matriz, copiamos la primera columna de la matriz de distancia, en la primera columna de la matriz solución.

Tras realizar el proceso de llenado de la matriz solución, se reconstruye el recorrido óptimo con la función. A continuación simplemente falta mostrar los resultados de la distancia, el recorrido, y el tiempo de ejecución:

```
Coste mínimo: 73.9876
Tiempo de ejecución: 0.118649

Solución:
1 38.24 20.42
16 39.36 19.56
12 38.47 15.13
13 38.15 15.35
14 37.51 15.17
6 37.56 12.19
7 38.42 13.11
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
8 37.52 20.44
4 36.26 23.12
2 39.57 26.15
3 40.56 25.32
1 38.24 20.42
```

Finalmente la gráfica del recorrido sería la siguiente:



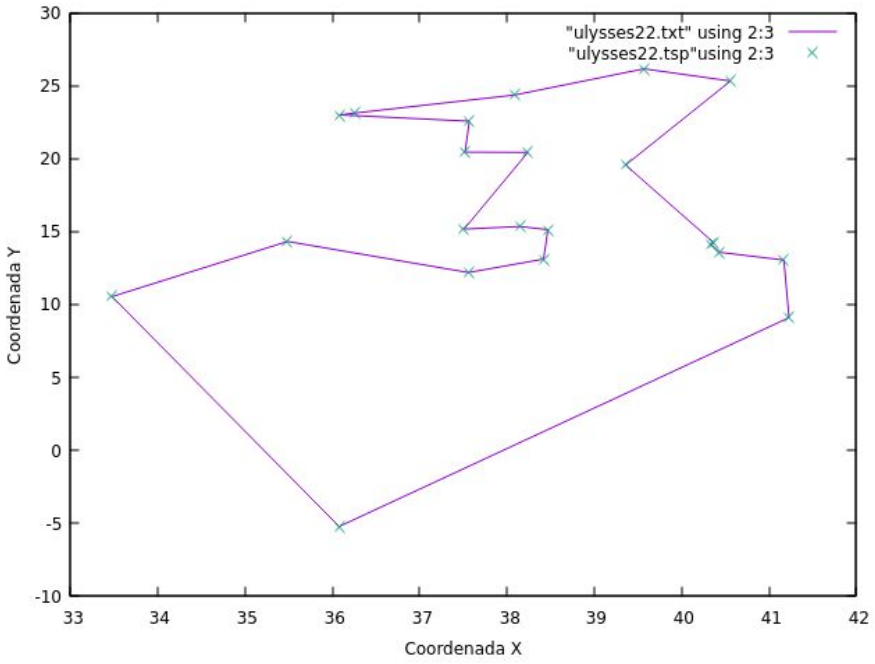
- **ULYSSES 22**

Procedemos a la ejecución nuevamente de algoritmo implementado con el conjunto de datos suministrado en el archivo **ulysses22.tsp**, el cual tiene 22 ciudades junto con sus coordenadas. En la función **tsp** inicializamos nuestra matriz solución a -1, para posteriormente llamar a la función **g** para que vaya rellenando ésta. Debido a las grandes dimensiones que presenta no mostramos captura de la matriz.

A continuación, llamamos a la función **recomponer_solucion** que será la que nos devuelva el camino recorrido. De forma que al ejecutar nuestro programa obtenemos como salida el coste mínimo, el recorrido de las ciudades junto con sus coordenadas y el tiempo de ejecución:

```
Coste mínimo: 75.3097
Solución:
1       38.24 20.42
8       37.52 20.44
22      37.57 22.56
18      36.09 23
4       36.26 23.12
17      38.09 24.36
2       39.57 26.15
3       40.56 25.32
16      39.36 19.56
21      40.37 14.23
20      40.33 14.15
19      40.44 13.57
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
12      38.47 15.13
13      38.15 15.35
14      37.51 15.17
1       38.24 20.42
Tiempo de ejecución: 27.9164
```

Si dibujamos el recorrido obtenido obtendremos el siguiente grafo:



5. Comparación programación dinámica - algoritmos voraces

Antes de proceder a comparar los diferentes resultados que hemos obtenido tanto en la práctica 3 con los algoritmos voraces, como en esta práctica con

programación dinámica, en primer lugar vamos a desarrollar la eficiencia que presenta nuestro algoritmo para resolver el TSP con programación dinámica:

Para calcular la eficiencia del nuestro algoritmo vamos a patir de las siguientes fórmulas:

·Si i ≠ 1, S no es vacío, S ≠ N - {1} y i ∉ S ==> $g(i,S) = \text{Min}_{j \in S} [L_{ij} + g(j, S - \{j\})]$

$g(i,\varnothing) = L_{i1}$, i = 2,3,...n

·Si S = Ø ===>

Donde Lij es la longitud del camino más corto del vértice i al vértice j y g(j, S - { j }) la distancia mínima que hay hasta j recorriendo todos los vértices de S pasando sólo una vez por cada uno. En primer lugar, para calcular g(j,Ø) hacemos n-1 consultas y para calcular g(i,S) hemos de hacerlo para todos los valores del cardinal de S tales que cumplan 1 ≤ |S| = k ≤ n-2 por lo que se deduce que tendríamos que hacer **(n-1)* Cn-2,k*k** sumas. De estas sumas tenemos que:

- n-1 son los valores que puede tomar la variable i
- k son los valores que puede tomar la variable j
- Cn-2,k el número de combinaciones de n-2 sobre k operaciones

Por tanto, si hacemos g(1, N-{1}) estaríamos haciendo n-1 sumas. De esta forma llegamos a que la eficiencia de nuestro algoritmo usando Programación Dinámica es:

$$O[2(n-1) + \sum_{k=1..(n-2)} (n-1)\times k\times C_{n-2,k}\ ] = O(n^2 2^n)$$

ya que

$$\sum_{k=1..r} k\times C_{r,k} = r2^{r-1}$$

Tras haber realizado ya el análisis de la eficiencia para nuestro algoritmo, procedemos a realizar una comparación entre los algoritmos. En primer lugar, en cuanto a lo eficiencia, podemos hablar que los algoritmos voraces en general son algoritmos rápidos, ya que su eficiencia varía entre O(n^2) (algoritmo cercanías y distancias) y O(n^3) (Enfoque inserción) y en general no han tenido problema para realizar las ejecuciones con todos los datos con los que hemos realizado pruebas, siempre dan tiempos de ejecución de la magnitud de milisegundos o segundos. El problema radica en que, aunque los algoritmos con enfoque greedy sean más rápidos que el algoritmo de PD (ya que tiene un orden de eficiencia mayor O(n^2*2^n), el PD asegura siempre devolver el resultado óptimo, cosa que los algoritmos voraces no consiguen. A parte de esto debemos destacar el hecho de que es inviable realizar la ejecución del algoritmo de PD para los datos att48 y a280,

debido a la absurda cantidad de requisitos que son necesarios tanto para almacenar las matrices como para realizar los cómputos. Esto por el lado de los algoritmos greedy es una ventaja puesto que al no usar memorización no disponen de ningún problema para realizar su tarea con cualquiera de los datos.

En segundo lugar vamos a observar los resultados de las distancias de los recorridos óptimos que han calculado cada uno de los algoritmos. Como podemos apreciar, y como ya sabíamos de antemano, el algoritmo usando PD siempre da la menor distancia, ya que dicho algoritmo siempre calcula el recorrido óptimo para cada conjunto de ciudades. Tras este le sigue en la mayoría de casos el de cercanías, distancia y por último inserción.

Por lo que podemos concretar que los algoritmos con un enfoque que utilice la programación dinámica, al siempre dar el resultado óptimo, siempre dará un mejor resultado que otro algoritmo con un enfoque greedy.

	ulysses 16	**ulysses 22**	**att48**	**a280**
cercanías	79	76	40503	3203
inserción	100	115	61949	6525
distancias	77	79	40138	3096
PD	73.9876	75.3097	-----	-----

Como conclusión al haber realizado estas dos prácticas, y al haber estudiado los dos enfoques (PD y greedy), podemos tener en claro una serie de conclusiones o de diferencias entre uno y otro enfoque:

PD	**Greedy**
Se progresa etapa por etapa con sub-problemas que se diferencian entre sí por sus tamaños	Se progresa etapa por etapa con subproblemas que no tienen por que coincidir en tamaño
Se generan muchas subsucesiones de decisiones	Solo se genera una sucesión de decisiones
Hay un gran uso de recursos (memoria)	La complejidad en tiempo suele ser baja (algoritmo relativamente rápidos)
En cada etapa siempre se compara los resultados con los precedentes. Siempre se obtiene la solución óptima	Como en cada etapa no se tiene en cuenta las decisiones precedentes, no hay garantía de obtener el óptimo