Práctica 4
Algoritmos Programación Dinámica
UGR - ETSIIT - ALGORÍTMICA - B1

Universidad de Granada

Componentes del Grupo "Oliva"

Jose Luis Pedraza Román

Pedro Checa Salmerón

Antonio Carlos Perea Parras

Raúl Del Pozo Moreno

Índice:

1. Descripción general de los problemas

En esta práctica vamos a analizar dos problemas mediante programación dinámica, el primer problema trata de encontrar la subsecuencia de caracteres más larga entre dos cadenas y el segundo problema que trata sobre el viajante de comercio.

En ambos problemas analizaremos la eficiencia y mostraremos casos de uso explicando su funcionamiento, explicando por qué la programación dinámica asegura un resultado óptimo.

2. Hardware y software utilizado

La compilación y ejecución de los algoritmos desarrollados en esta práctica se han realizado en un sistema Linux con la distribución “Arch Linux” de 64 bits, con un procesador i7-4510U a 2GHz.

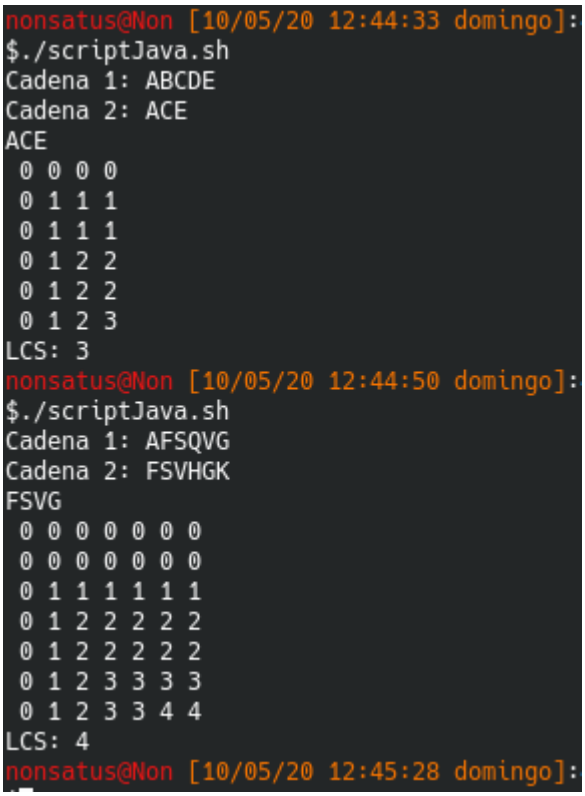
```
$uname -rms
Linux 5.5.8-arch1-1 x86_64
```

3. Subsecuencia de caracteres más larga

En este problema buscaremos la subsecuencia de caracteres (no consecutiva) más larga entre dos cadenas.

Para ello se usa una matriz en la que almacenaremos el número de coincidencias respecto a las coincidencias anteriores, de forma que, al comprobar dos caracteres, iguales o no, arrastrará el número de coincidencias anterior.

En la siguiente imagen podemos ver dos ejemplos de ejecución correctos:



En la primera ejecución se comparan las cadenas “ABCDEFGA” y “ACDFGA”, visualmente, se puede ver rápidamente que la solución es: “ACDFGA” con una longitud de 6 caracteres.

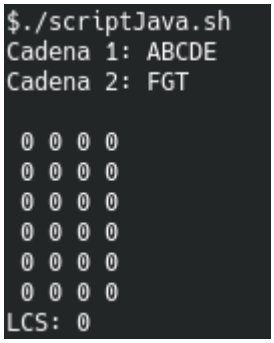
En la segunda ejecución se comparan las cadenas “ABCDEFGA” y “ACADE”, cuya solución es “ACDE” con una longitud de 4 caracteres.

Para obtener dichos resultados se utiliza una matriz donde se almacenan las coincidencias de caracteres.

La primera fila y columna de 0 corresponde con la comparación de cada uno de los caracteres con un carácter nulo, de forma que inicializa el número de coincidencias a 0.

Así, para cada carácter comprobado, tiene en cuenta las coincidencias anteriores.

En la imagen anterior podemos ver un ejemplo de ejecución en el que no ha encontrado ninguna coincidencia, por lo que su matriz solución es 0.



Hay que discernir dos casos:

- Los dos caracteres son iguales

En este caso, el número de coincidencia aumenta en 1, sumado al número de coincidencias que hubiera antes, para ello, tiene en cuenta el valor de la matriz correspondiente a la fila y columna anterior (diagonal superior izquierda).

- Los dos caracteres son distintos

En este caso, el número de coincidencia no aumenta en 1; sin embargo, mantiene el máximo valor de coincidencias que hubiera en la fila o columna anterior.

A continuación, se va a mostrar un ejemplo práctico:

Cadena 1: “ABCD”
Cadena 2: “ACD”

Paso 1: Comparar cada carácter con un carácter vacío, como no hay coincidencia, la primera fila y columna está inicializada a 0.

	-	**A**	**C**	**D**
-	0	0	0	0
A	0			
B	0			
C	0			
D	0			

Paso 2: Comprueba para cada carácter de la cadena 1, con cada carácter de la cadena 2 (todos con todos), y dependiendo de si los caracteres coinciden, aplicamos la metodología explicada anteriormente del caso correspondiente (explicado anteriormente).

“A” con “A” coincide así que suma 1 al valor de la celda superior izquierda con valor 0.

	-	A	C	D
-	0	0	0	0
A	0	1		
B	0			
C	0			
D	0			

“A” no coincide con “C” ni “D” así que mantiene el máximo de la celda de la fila y columna anterior, en este caso, va a arrastrando el 1 de la fila, ya que el Max (0,1) = 1. En azul claro se ve que celdas utiliza para obtener el máximo (“AA” y “AC”), en verde se indica la celda donde inserta el valor.

	-	A	C	D
-	0	0	0	0
A	0	1	1	
B	0			
C	0			
D	0			

	-	A	C	D
-	0	0	0	0
A	0	1	1	1
B	0			
C	0			
D	0			

Ya se ha terminado de comparar el carácter “A” de la cadena 1 y se pasa a comparar el segundo carácter de la cadena 1, que es “B”.

	-	A	C	D
-	0	0	0	0
A	0	1	1	1
B	0	1	1	1
C	0			
D	0			

Ahora compara “B” con “A”, “C” y “D”, como ninguno coincide mantiene el máximo de las celdas correspondientes a cada comparación.

Para la fila con el carácter “C”, se repite el proceso hasta llegar a la columna “C”, como hay coincidencia, en vez de mantener el máximo de la fila y columna anterior, obtiene el valor de la celda superior izquierda y suma 1.

	-	A	C	D
-	0	0	0	0
A	0	1	1	1
B	0	1	1	1
C	0	1	2	
D	0			

Este proceso se repite hasta completar la matriz

	-	A	C	D
-	0	0	0	0
A	0	1	1	1
B	0	1	1	1
C	0	1	2	2
D	0			

	-	A	C	D
-	0	0	0	0
A	0	1	1	1
B	0	1	1	1
C	0	1	2	2
D	0	1		

	-	A	C	D
-	0	0	0	0
A	0	1	1	1
B	0	1	1	1
C	0	1	2	2
D	0	1	2	

	-	A	C	D
-	0	0	0	0
A	0	1	1	1
B	0	1	1	1
C	0	1	2	2
D	0	1	2	3

Paralelamente, se rellena una matriz en la que en cada celda se indica la celda desde la cual proviene, esto sirve para recorrer de una forma más eficiente la matriz para mostrar la cadena, ya que, si no se usara, habría que realizar 6 accesos en la matriz para decidir el siguiente carácter solución, usando una matriz indicando la dirección, solamente se realiza 1 acceso por carácter.

Empezando desde la última celda (celda esquina inferior derecha “DD”), obtiene el valor de la dirección y recorre paralelamente la matriz solución hasta encontrar un 0, en este caso, la ruta seria: D → D → T → D → 0, que en la matriz solución corresponde con “ACD”

Cuando se detecta una “diagonal”, la celda corresponde con un carácter solución, si la celda almacena un “top” o “left”, el carácter no es solución y se continúa el camino.

	-	A	C	D
-	0	0	0	0
A	0	D	L	L
B	0	T	T	T
C	0	T	D	L
D	0	T	T	D

4. Viajante de comercio

En este apartado vamos a calcular la ruta óptima de una serie de ciudades mediante programación dinámica.

Dado un grafo ponderado de valores no negativos asociados a sus arcos, queremos encontrar el circuito más corto posible que comience y termine en un mismo nodo; es decir queremos encontrar un camino cerrado que recorra todos los nodos una y solo una vez y que tenga longitud mínima.

$$g(i,S) = Min_{j \epsilon S} [L_{i\,j} + g(j, S - \{j\})]$$

Donde definimos el valor $g(i,S)$ para cada nodo i, como la longitud del camino más corto desde el nodo i al nodo 1 que pasa exactamente una vez a través de cada nodo de S.

Si suponemos que el circuito comienza y termina por el nodo 1, la ecuación nos queda de la siguiente forma:

$$g(1, N - \{1\}) = Min_{2<=j<=n} [L_{1\,j} + g(j, N - \{1, j\})]$$

Aquí, $g(1, N - \{1\})$corresponde a la longitud del circuito óptimo Hamiltoniano, que es lo que estamos buscando.

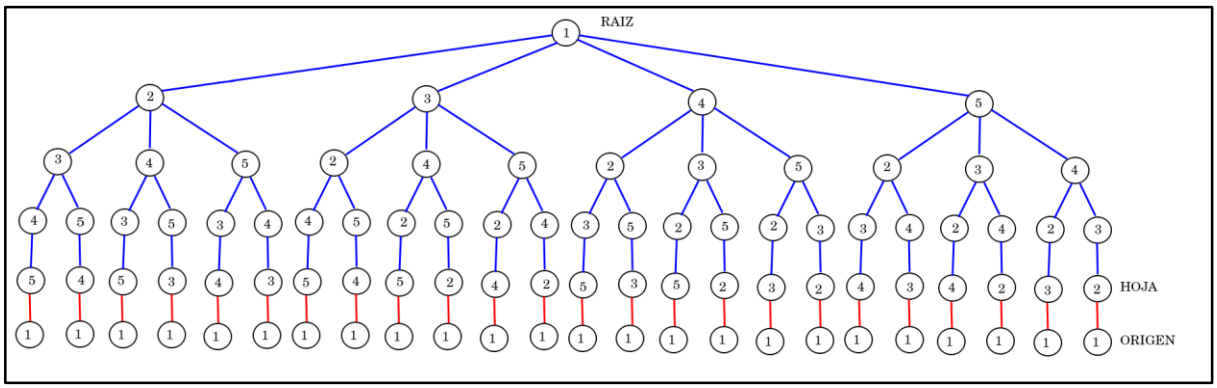
En otros términos; para calcular el circuito óptimo, necesitamos calcular g (i, S), primero para las distancias directas (aquellas que no pasan por vértice alguno), seguir calculando para todos los conjuntos S que solo contienen un nodo, calculando g (i, S) a partir de la ecuación mostrada anteriormente; y continuamos aumentando el número de nodos de S y calculando g (i, S) a partir de los datos anteriores hasta llegar al nodo 1, que incluirá en S todos los demás nodos.

Para tener en cuenta todos los distintos circuitos que se pueden dar, representamos los nodos estructurados en forma de árbol, donde cada nodo tiene como nodos hijos todos aquellos nodos que no se encuentren por encima suya, ya que aún no se han visitado.

De esta forma, aunque el nodo raíz siempre sea el nodo 1, se representan todos los posibles circuitos Hamiltonianos que se podrían tomar, por ejemplo, todos los siguientes circuitos empezando por diferente ciudad, corresponden a la misma ruta:

Circuito: { 1 2 3 4 5 1 } = { 2 3 4 5 1 2 } = { 3 4 5 1 2 3 } = { 4 5 1 2 3 4 } = { 5 1 2 3 4 5 }

Así, un árbol para 5 nodos quedaría de la siguiente forma:



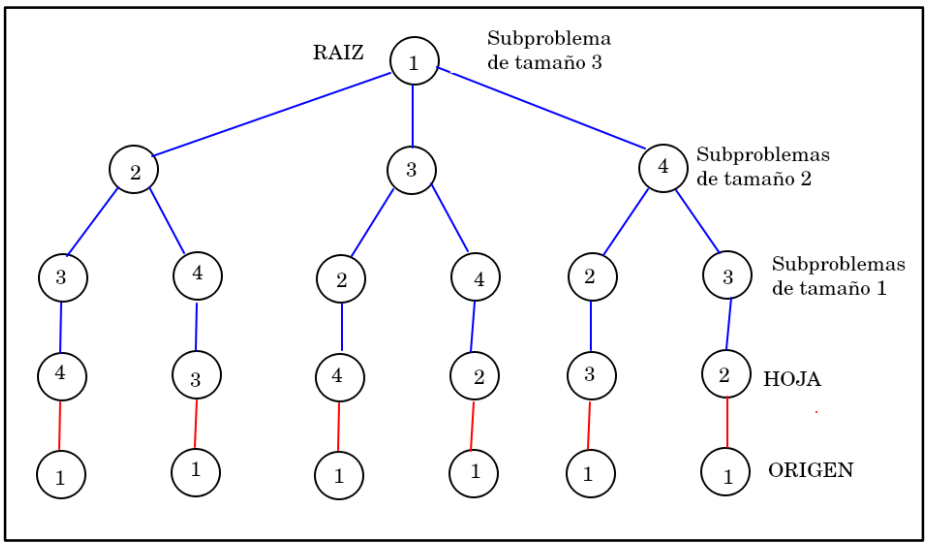
En nuestro programa esto lo logramos con una función recursiva que, empezando por el nodo raíz (el 1), hace lo siguiente:

- Si nos encontramos con un nodo hoja, devolvemos un entero con la distancia desde ese nodo al nodo raíz (la distancia de regreso a la ciudad origen).
- Si nos encontramos con un nodo que NO sea hoja:
    - Llamamos a tantas funciones recursivas como hijos tenga el nodo.
    - Recogemos las distancias devueltas por estas funciones y le sumamos a cada una la distancia desde el nodo hijo correspondiente al nodo actual (padre).
    - Nos quedamos con el menor de todos estos valores y lo devolvemos.

4.1. Características programación dinámica

Naturaleza n-etápica del problema

El viajante de comercio es de naturaleza n-etápica debido a que el problema va dividiéndose en subproblemas que se diferencian en una unidad de tamaño, es decir, para una ruta de 4 ciudades, el problema se divide en un subproblema de 3 ciudades, luego en subproblemas de 2 ciudades, luego en subproblemas de 1 ciudad (nodo hoja).



Verificación del POB

Una política óptima tiene la propiedad de que cualquiera que sea el estado inicial y la decisión inicial, las decisiones restantes deben constituir una política óptima en relación con el estado resultante de la primera decisión. Una política óptima sólo puede estar formada por sub-políticas óptimas.

Este problema verifica el Principio del Óptimo de Bellman porque en cada etapa, las decisiones a tomar en las siguientes etapas son independientes a la etapa actual, es decir, la distancia que hay entre, por ejemplo, la ciudad 3 y la ciudad 4 (independientemente de las ciudades que pueda haber en medio) no va a afectar a la distancia que hay entre la ciudad 3 y las ciudades predecesoras.

Planteamiento de una recurrencia

Cada vez que se visita una ciudad, quedan N-i ciudades por visitar, donde i es el número de ciudades ya visitadas, por tanto, existen N-i posibilidades desde la última ciudad visitada, la forma de comprobar todas estas posibilidades es realizar una recurrencia para cada ciudad restante, la cual retorna la distancia acumulada desde esa ciudad a la ciudad retorno (origen).

Siendo:

- k → clave que indica las ciudades restantes
- hijo → nodo sobre el que se hace la recursión
- raíz → nodo inicial del árbol (ciudad inicial)
- hoja → nodo final (última ciudad por visitar)
- i → nodo hijo del nodo al que se le está haciendo la recursión

$$distancia\ (hijo, raíz) \qquad => hijo = hoja$$

$$recurrencia\ (k, hijo) = distancia\ (hijo, i) + min(recurrencia\ (k - 1, i)) \qquad => hijo\ != hoja$$

Ejemplo para 4 ciudades:

Ciudades visitadas acumuladas						Ciudades restantes
1						N=3
1 2		1 3		1 4		N=2
1 2 3	1 2 4	1 3 2	1 3 4	1 4 2	1 4 3	N=1
1 2 3 4	1 2 4 3	1 3 2 4	1 3 4 2	1 4 2 3	1 4 3 2	N=0

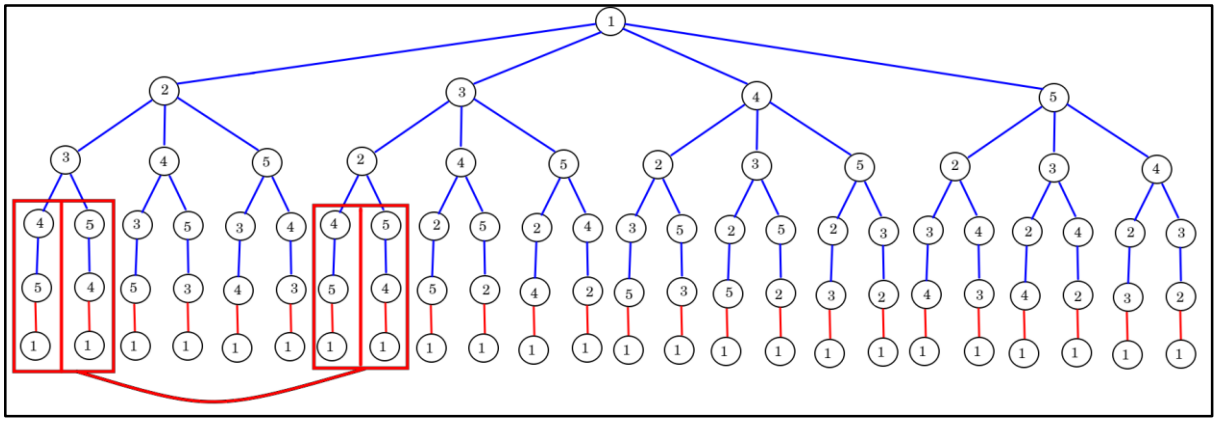
Cálculo de la solución (enfoque adelantado o atrasado)

Para evitar tener que calcular las distancias de una ciudad a otra cada vez, utilizamos una matriz donde almacenaremos las distancias entre cada par de ciudades y así solamente tendremos que realizar un acceso de memoria para obtener la distancia.

	A	B	C	D
A	0	6	1	3
B	6	0	4	7
C	1	4	0	5
D	3	7	5	0

Para evitar calcular recursivamente secuencias de ciudades ya calculadas, se necesita una estructura donde almacenar dichas operaciones, para que en el caso de que se encuentre una secuencia previamente realizada, se pueda obtener el valor ya calculado.

Por ejemplo, el siguiente árbol evitará volver a calcular el bloque rojo de la derecha ya que anteriormente hemos obtenido la distancia mínima para dicho bloque:



En la imagen superior, la primera vez que se visita el subárbol con nodo padre 3, se creará un unordered_map que representaría los nodos hijos 4 y 5, almacenando la distancia del nodo 4 hasta el nodo origen (distancia {4 → 5 → 1}) y la distancia desde el nodo 5 hasta el nodo origen (distancia {5 → 4 → 1}).

Así, cuando recursivamente nos encontremos que el unordered_map ya está creado, como pasa al llegar al nodo 2 (cuadro rojo más a la derecha), en vez de hacer las recursiones de nuevo, calcula la distancia desde cada nodo hijo al padre sumándole la distancia ya calculada del nodo hijo correspondiente, quedándose con la mejor distancia.

Si se tiene que:

distancia (2, 4) = 4	unordered_map[clave][0] = 9	Distancia nodo 2 al nodo {4, …, 1} = 4 + 9 = 13
distancia (2, 5) = 8	unordered_map[clave][1] = 3	Distancia nodo 2 al nodo {5, …, 1} = 8 + 3 = 11

Por ejemplo, aunque la mejor distancia desde el nodo padre al hijo seria al nodo 4, prefiere la distancia al nodo 5 ya que globalmente, la distancia es mejor.

Siendo la ruta más óptima en este momento la ruta por el nodo 2 → 5 → … → 1

En el caso de que la llamada recursiva detecte que el nodo es un nodo hoja (última ciudad por visitar) devuelve la distancia desde el nodo hoja (última ciudad sin visitar) hasta el nodo origen (nodo raíz)

Para ello usamos un unordered_map que almacena mediante una clave potencia de 2 (binaria) que almacena las distancias óptimas para cada nodo representado en su clave. (unordered_map<int, vector<int>>)

Por ejemplo:

Clave: 14 (en binario 1110) representa las ciudades {4, 3, 2}, almacenando un vector de distancias de tamaño 3, donde la primera posición del vector almacena la distancia óptima para el nodo 4, la segunda posición almacena para la ciudad 3 y la última posición para la ciudad 2.

La representación de las claves para las posibles combinaciones serían las siguientes, de las cuales todas no siempre van a ser usadas, sino que se van a crear según se calculen.

Si se tuvieran 4 ciudades, se tendría una clave máxima 2^N - 1 = 1111, donde se representan todas las ciudades, pero al empezar siempre por la ciudad 1, esta ciudad se sustrae de la clave quedando (2^N - 1) - 1.

2^16 = pow (2, 4) = 16 = 1 0000
(16) 1 0000 - 1 = (15) 1111 [se obtiene el Max]
(15) 1111 - 1 = (14) 1110 [se sustrae la ciudad 1 (la ciudad inicial)]

Dec	Bin	Ciudades		Dec	Bin	Ciudades
0	0000	-		8	1000	{4}
1	0001	{1}		9	1001	{1, 4}
2	0010	{2}		10	1010	{2, 4}
3	0011	{1, 2}		11	1011	{1, 2, 4}
4	0100	{3}		12	1100	{3, 4}
5	0101	{1, 3}		13	1101	{1, 3, 4}
6	0110	{2, 3}		14	1110	{2, 3, 4}
7	0111	{1, 2, 3}		15	1111	{1, 2, 3, 4}

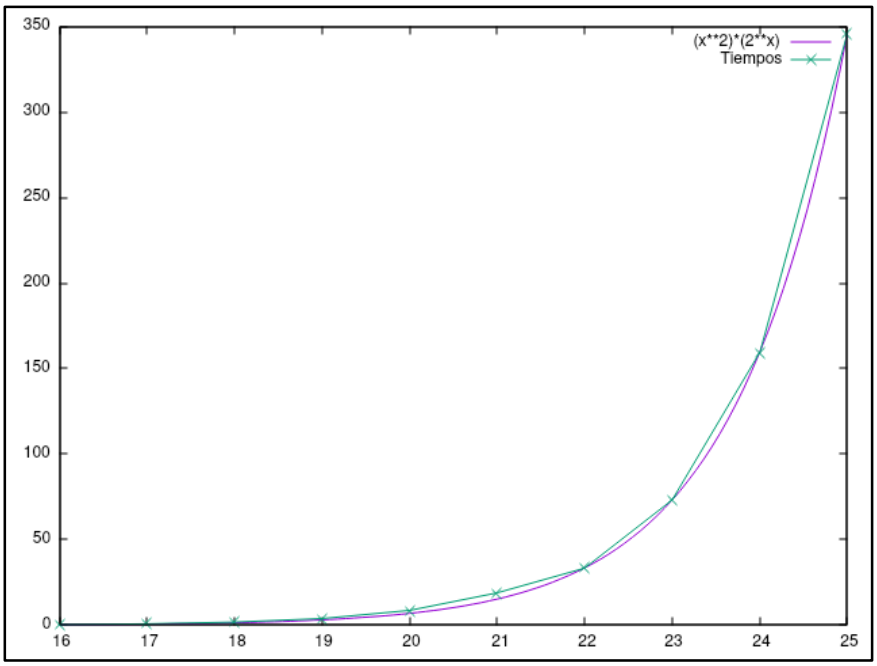
4.2. Resultados

A continuación, se puede ver una tabla con algunos tiempos de ejecución obtenidos para distinto número de ciudades, debido a que este tipo de algoritmo tiene el problema de que a más ciudades haya, más tiempo va a tardar. En este tipo de problema, si se resolviera por fuerza bruta, tendría una eficiencia $O(n!)$, mientras que mediante programación dinámica, se obtiene una eficiencia $O(n^2 2^n)$.

Es mucho más rápido, pero sigue teniendo un crecimiento muy rápido para cada ciudad añadida, tal y como se puede ver en la siguiente tabla donde hay, con una diferencia de 9 ciudades, un incremento de 345 segundos.

Archivo	Número ciudades	Tiempo(s)	Distancia óptima
ulysses16	16	0.287197	71
ulysses22	17	0.797242	71
	18	1.83127	71
	19	3.91046	71
	20	8.6741	72
	21	18.9258	72
	22	33.3153	72
att48	23	73.195	24483
	24	159.232	24688
	25	345.821	24928

A la derecha vemos cómo según la eficiencia teórica que tiene la programación dinámica para el viajante de comercio, se ajusta correctamente a los tiempos obtenidos:



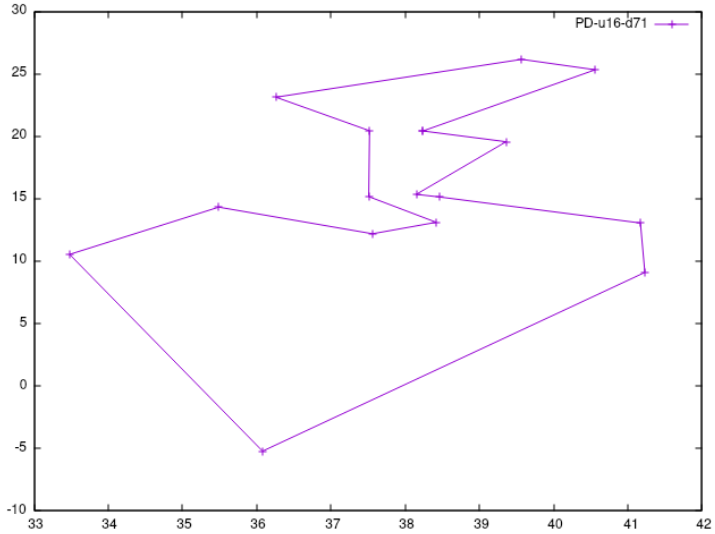
Como se puede ver, al realizar un ajuste híbrido de los tiempos obtenidos a la función $n^2 2^n$vemos que el error obtenido es muy bajo (0.3574%).

```
gnuplot> f(x) = a0*(x**2)*(2**x)
gnuplot> fit f(x) "data/gnuplot/tiempos.dat" via a0
iter      chisq       delta/lim  lambda   a0
   0 5.5808042040e+20   0.00e+00  7.47e+09    1.000000e+00
   1 4.6122348794e+18  -1.20e+07  7.47e+08    9.090911e-02
   2 4.6030242268e+12  -1.00e+11  7.47e+07    9.083477e-05
   3 4.7775339704e+02  -9.63e+14  7.47e+06    1.740341e-08
   4 1.7460191841e+01  -2.64e+06  7.47e+05    1.649524e-08
   5 1.7460191841e+01  -2.64e-08  7.47e+04    1.649524e-08
iter      chisq       delta/lim  lambda   a0

After 5 iterations the fit converged.
final sum of squares of residuals : 17.4602
rel. change during last iteration : -2.63704e-13

degrees of freedom    (FIT_NDF)                        : 9
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 1.39285
variance of residuals (reduced chisquare) = WSSR/ndf   : 1.94002

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0              = 1.64952e-08      +/- 5.896e-11    (0.3574%)
gnuplot> plot f(x) title "(x**2)*(2**x)" w l, "data/gnuplot/tiempos.dat" title "Tiempos" w lp
gnuplot>
```

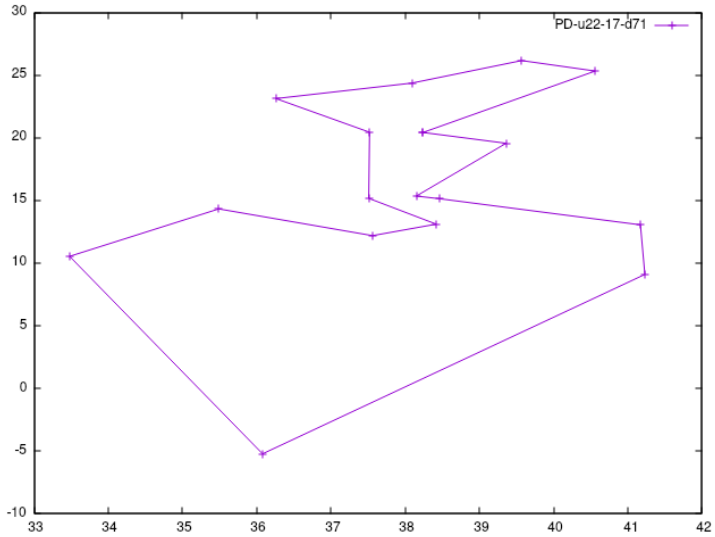
Para terminar, vamos a mostrar las rutas obtenidas en las ejecuciones de nuestro programa:

```
nonsatus@Non [18/05/20 0:25:53 lunes]:~/Documents/UGR1920/ALG/Practicas/P4_PD/
$make
g++ -std=gnu++0x tsp_pd.cpp -o tsp_pd
nonsatus@Non [18/05/20 0:25:59 lunes]:~/Documents/UGR1920/ALG/Practicas/P4_PD/
$./tsp_pd data/tsp/ulysses16.tsp
Distancia: 71
Ruta: 1 3 2 4 8 14 7 6 15 5 11 9 10 12 13 16 1
Tiempo: 0.287197
nonsatus@Non [18/05/20 0:26:02 lunes]:~/Documents/UGR1920/ALG/Practicas/P4_PD/
$./tsp_pd data/tsp/ulysses22.tsp
Distancia: 72
Ruta: 1 3 2 17 18 4 22 8 12 13 14 7 6 15 5 11 9 10 19 20 21 16 1
Tiempo: 33.3153
nonsatus@Non [18/05/20 0:28:58 lunes]:~/Documents/UGR1920/ALG/Practicas/P4_PD/
$./tsp_pd data/tsp/att48_23.tsp
Distancia: 24483
Ruta: 1 8 9 15 18 7 6 19 17 20 12 11 13 21 10 4 2 5 14 23 3 22 16 1
Tiempo: 73.195
nonsatus@Non [18/05/20 0:30:22 lunes]:~/Documents/UGR1920/ALG/Practicas/P4_PD/
$./tsp_pd data/tsp/att48_24.tsp
Distancia: 24688
Ruta: 1 8 9 15 18 7 6 19 17 20 12 11 13 21 24 10 4 2 5 14 23 3 22 16 1
Tiempo: 159.232
nonsatus@Non [18/05/20 0:33:09 lunes]:~/Documents/UGR1920/ALG/Practicas/P4_PD/
$./tsp_pd data/tsp/att48_25.tsp
Distancia: 24928
Ruta: 1 8 9 15 18 7 6 19 17 20 12 11 13 21 24 10 4 2 5 25 14 23 3 22 16 1
Tiempo: 345.821
```

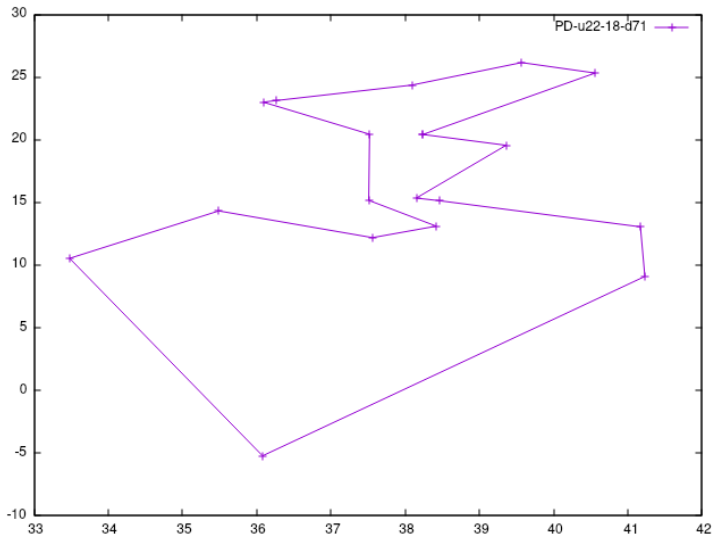
```
nonsatus@Non [18/05/20 16:36:18 lunes]:~/Documents/UGR1920/ALG/
$./tsp_pd data/tsp/ulysses22_17.tsp
Distancia: 71
Ruta: 1 3 2 17 4 8 14 7 6 15 5 11 9 10 12 13 16 1
Tiempo: 0.797242
nonsatus@Non [18/05/20 16:36:23 lunes]:~/Documents/UGR1920/ALG/
$./tsp_pd data/tsp/ulysses22_18.tsp
Distancia: 71
Ruta: 1 3 2 17 4 18 8 14 7 6 15 5 11 9 10 12 13 16 1
Tiempo: 1.83127
nonsatus@Non [18/05/20 16:36:27 lunes]:~/Documents/UGR1920/ALG/
$./tsp_pd data/tsp/ulysses22_19.tsp
Distancia: 71
Ruta: 1 3 2 17 4 18 8 14 15 5 11 9 10 19 6 7 12 13 16 1
Tiempo: 3.91046
nonsatus@Non [18/05/20 16:36:35 lunes]:~/Documents/UGR1920/ALG/
$./tsp_pd data/tsp/ulysses22_20.tsp
Distancia: 72
Ruta: 1 3 2 17 4 18 8 12 13 14 7 6 15 5 11 9 10 19 20 16 1
Tiempo: 8.6741
nonsatus@Non [18/05/20 16:36:46 lunes]:~/Documents/UGR1920/ALG/
$./tsp_pd data/tsp/ulysses22_21.tsp
Distancia: 72
Ruta: 1 3 2 17 4 18 8 12 13 14 7 6 15 5 11 9 10 19 20 21 16 1
Tiempo: 18.9258
```



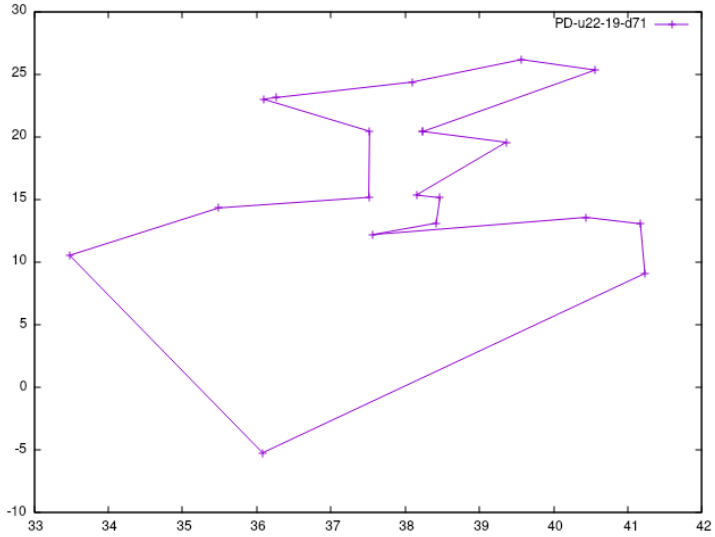
Ulysses16 con todas sus ciudades



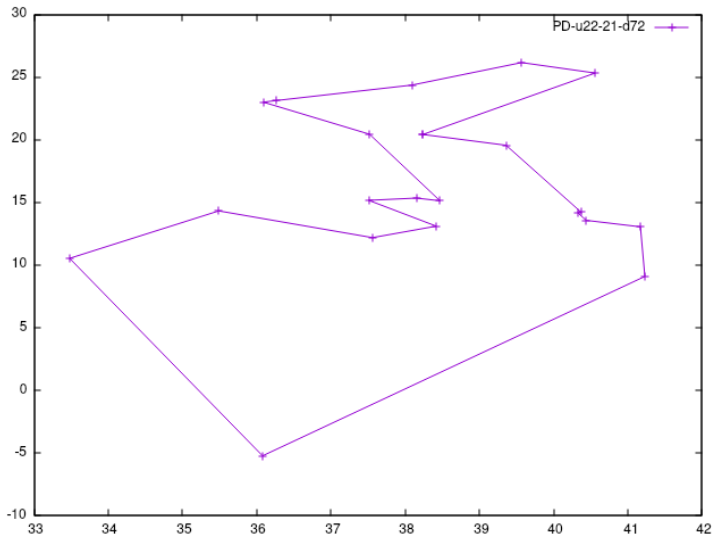
Ulysses22 con 17 ciudades



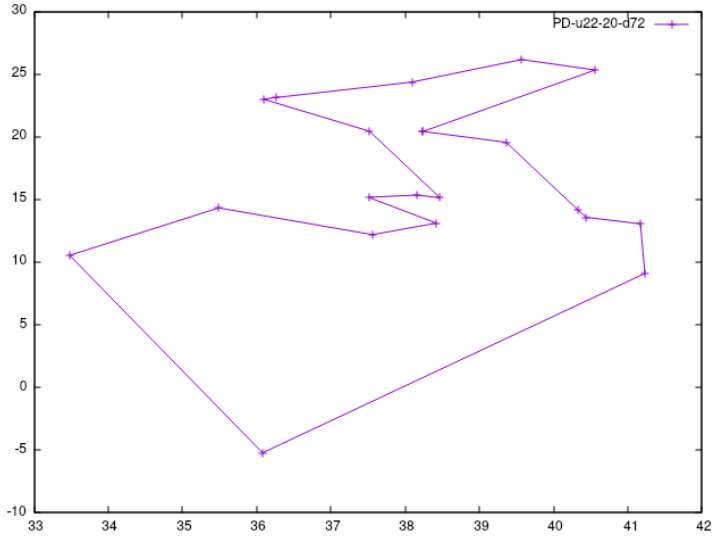
Ulysses22 con 18 ciudades



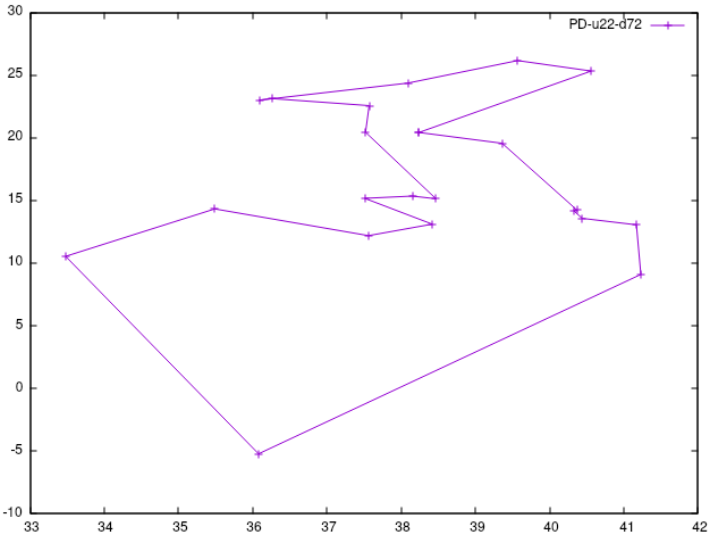
Ulysses22 con 19 ciudades



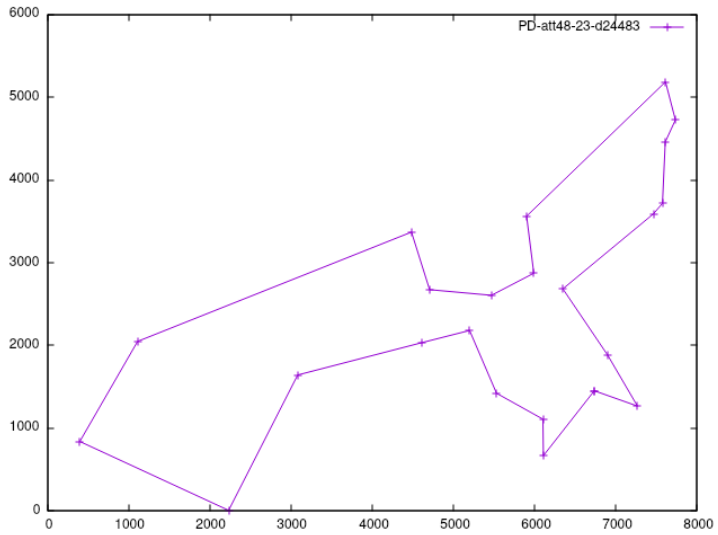
Ulysses22 con 20 ciudades



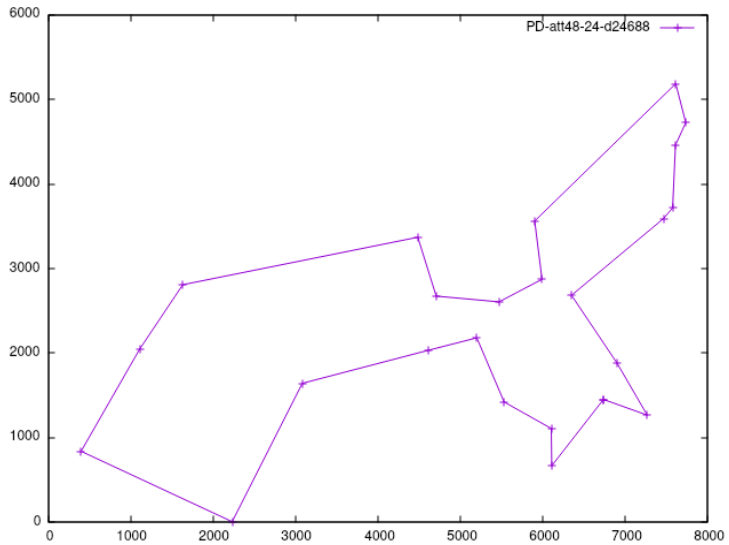
Ulysses22 con 21 ciudades



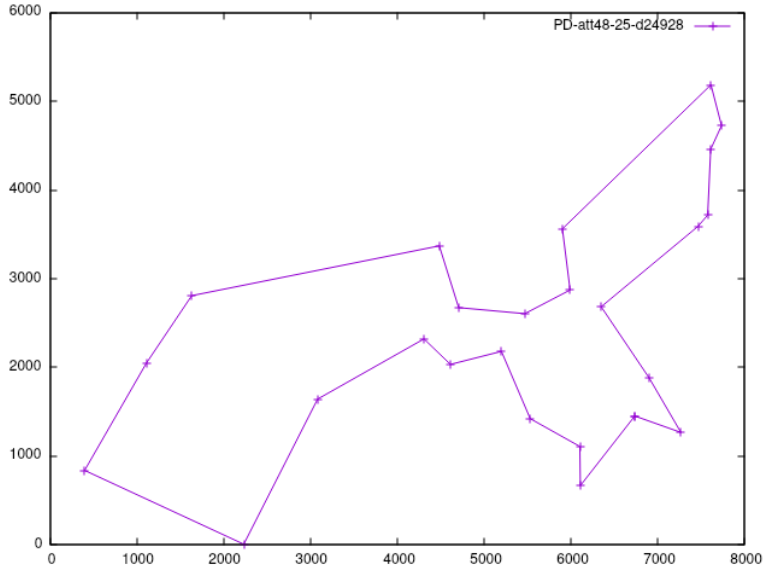
Ulysses22 con todas sus ciudades



Att48 con 23 ciudades



Att48 con 24 ciudades



Att48 con 25 ciudades

5. Conclusiones

Al estudiar programación dinámica para los dos ejercicios propuestos (subsecuencia de caracteres y viajante de comercio) hemos visto la importancia de elegir una correcta estructura de datos con la que almacenar soluciones y así evitar recalcular operaciones ya realizadas de forma que podemos obtener una solución eficiente (óptima) para un problema en un espacio de tiempo variable, ya que este variará del tipo de problema a resolver.

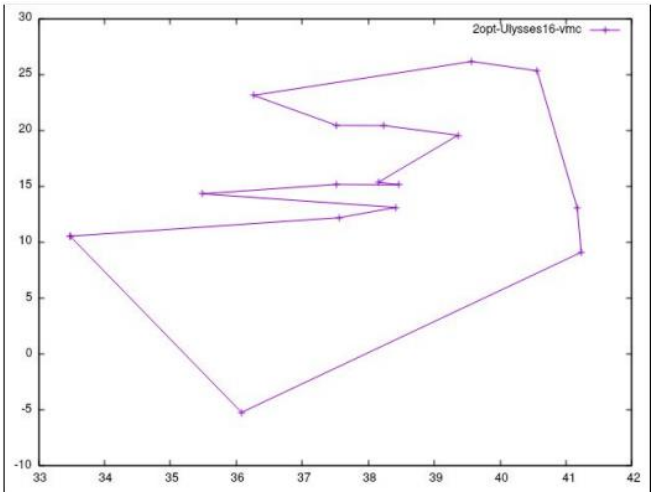
Así, comparando técnicas previas como algoritmos greedy, podemos observar que los algoritmos greedy nos permiten obtener una solución bastante aproximada con una relación tamaño-tiempo alta, es decir, con algoritmos greedy obtuvimos soluciones para problemas con 280 ciudades teníamos una solución en 1.05 segundos, mientras que, con programación dinámica, con 25 ciudades el tiempo de ejecución sube hasta 5 minutos y 45 segundos, pero permitiendo obtener la ruta óptima.

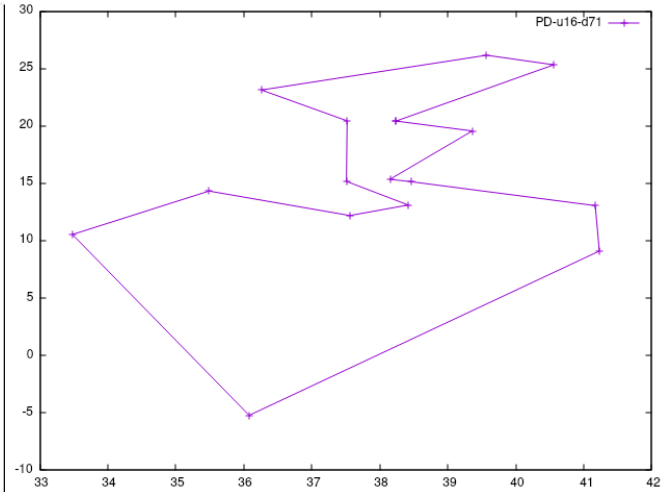
Programación Dinámica	Enfoque Greedy
Se progresa etapa por etapa con sub-problemas que se diferencian entre sí en una unidad en sus tamaños.	Se progresa etapa por etapa con sub-problemas que no tienen por qué coincidir en tamaño.
Se generan muchas sub-sucesiones de decisiones.	Solo se genera una sucesión de decisiones.
Hay un gran uso de recursos (memoria).	La complejidad en tiempo suele ser baja.
En cada etapa se comparan los resultados obtenidos, con los precedentes: Siempre obtienen la solución óptima.	Como en cada etapa se selecciona lo mejor de lo posible sin tener en cuenta las decisiones precedentes, no hay garantía de obtener el óptimo.

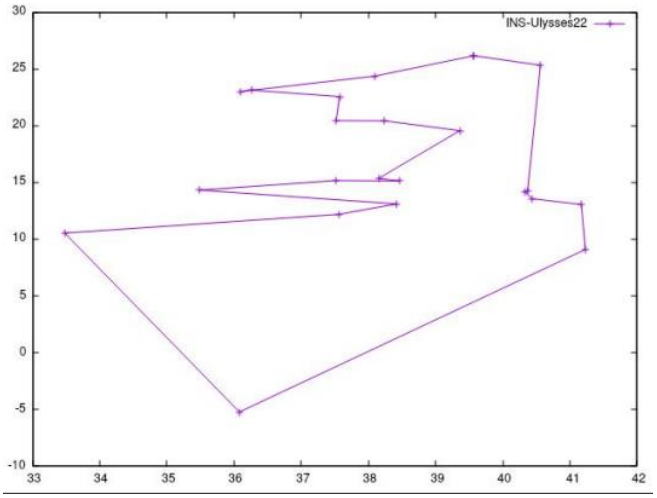
A continuación, vemos la ruta gráfica para ulysses16 y ulysses22 obtenidos en greedy (usando la ruta más óptima de las tres heurísticas realizadas en greedy) y en programación dinámica:

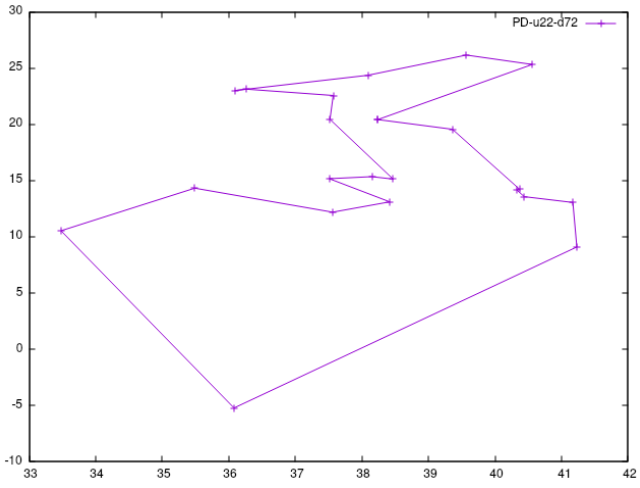
Fichero	Distancia		Tiempo	
	PD	Greedy	PD	Greedy
Ulysses16	71	107	0.287197	0.00270537
Ulysses22	72	84	33.3153	0.000489116

Podemos ver que los algoritmos Greedy tardan bastante menos que la Programación Dinámica, pero ésta última nos asegura encontrar siempre el recorrido óptimo mientras que con Greedy por norma general no lo obtendremos.









6. Bibliografía

Lecciones de Algorítmica - Verdegay Galdeano, José Luis
https://drive.google.com/drive/folders/1LDuRCkdOhjZFDG6xjq4m94_Q8GBvqCRx
https://www.youtube.com/watch?v=Q4zHb-Swzro