Universidad De Granada

Práctica 4
Programación Dinámica

Víctor José Rubia López
B3
Fecha de entrega 18/05/2020

Contenido

Capítulo 1: Programación dinámica

1.1. Descripción general

La programación dinámica es una técnica que resuelve problemas de forma recursiva consiguiendo así una mejora en la eficiencia.

Normalmente, con recursividad resolvemos los subproblemas de forma repetida, sin embargo, en programación dinámica guardamos la solución de estos subproblemas de forma que no tengamos que resolverlos de nuevo. Esto es lo que se conoce como *Memoization* (del inglés).

Capítulo 2: Problema del viajante de comercio

2.1. Problema del viajante de comercio usando Programación Dinámica

Al igual que en la práctica anterior, el problema del TSP (Travelling Salesman Problem), consiste en un conjunto de ciudades y distancias hacia un número finito de ciudades. Su objetivo es visitar todas las ciudades del mapa como máximo una vez y regresar al punto de partida.

Para abordarlo mediante el uso de Programación Dinámica consideramos un conjunto de vértices {1, 2, 3, 4, ..., n} y que empezamos y terminamos en el punto 1. Entonces, para cada vértice i distinto de 1, trataremos de encontrar el camino con coste mínimo con la ciudad 1 como punto de partida, i como punto de finalización y pasando por cada vértice una única vez. Siendo el coste de un camino *coste(i)*, el coste de su ciclo sería *coste(i) + distancia(i, 1)*, donde *distancia(i, 1)* es la distancia de i a 1. Para terminar, devolveríamos el mínimo de todos los *[coste(i) + distancia(i, 1)]*.

Para calcular el *coste(i)* usando Programación Dinámica, necesitamos encontrar una similitud entre los subproblemas para poder realizarlos de forma recursiva. Definimos el término *C(S, i)* como el mínimo coste del recorrido visitando cada vértice en el conjunto S una única vez, empezando en el vértice 1 y terminando en el vértice i.

Empezaremos con todos los subconjuntos de tamaño 2 y calcularemos *C(S, i)* para todos ellos, donde S es el subconjunto de vértices. Tras ello, calcularemos *C(S, i)* para todos los subconjuntos de tamaño 3 y así repetidas veces. Subrayamos que el vértice 1 debe estar presente en cada subconjunto.

Para un conjunto de vértices *n*, consideramos *n-2* subconjuntos, cada uno de tamaño *n-1*, de modo que no tengan *n* contenido. Usando esta explicación podemos escribir lo siguiente:

```
If tamaño de S es 2, entonces S debe ser {1, i},
 C(S, i) = distancia(1, i)
Else if tamaño de S es mayor que 2
 C(S, i) = mínimo { C(S-{i}, j) + distancia(j, i)} donde j pertenece a S, j
```

Con esto, calculamos que hay un máximo de $O(n \cdot 2^n)$ subproblemas y cada uno tarda un tiempo lineal en resolverse, por lo que el tiempo total teórico sería $O(n^2 \cdot 2^n)$. Su coste en espacio para conservar C y j es también exponencial. Tenemos en cuenta que este orden de eficiencia es mejor que la $O(n!)$ que obtenemos con la estrategia de fuerza bruta. Por ello, este método de resolución es inviable cuando tenemos un gran número de vértices

A continuación, expongo el pseudocódigo del algoritmo que he diseñado:

2.1.1. Pseudocódigo

```
función algoritmo_pd(bitmask, actual, subproblema, distancias) devuelve sol
variables camino, resultado,i:sol
principio
      si bitmask == (1 << distancias.size-1 entonces devuelve par(camino,
distancias[actual][0])

      si subproblema[bitmask-1][actual].second != -1
                  entonces devuelve subproblema[bitmask-1, actual]

      variable min:= INT_MAX

      para todo j en distancias.size hacer

            variable res:=bitmask & (1 << i)
            si res == 0
                  entonces
                        variable nuevoBitmask:= bitmask | (1 << i)
                        variable nuevoPar:= algoritmo_pd(nuevoBitmask, i, subproblema,
distancias)
                        variable nuevoMin:= distancias[actual][i] + nuevoPar.second
                        variable nuevoPath(nuevoPar.first)
                        nuevoPath.push_back(actual)

                        si nuevoMin < min
                              entonces
                                    min:= nuevoMin
                                    camino:= nuevoPath
                        fsi
            fsi
      fpara

      subproblema[bitmask-1][actual].second:= min
      subproblema[bitmask-1][actual].first:= camino

      devuelve sol:=make_pair(camino, min);
fin
```

La forma en la que he diseñado el algoritmo necesita rastrear las ciudades ya visitadas junto con la última ciudad visitada para que así podamos identificar cada estado del problema inequívocamente.

Por lo tanto, identificamos dicho estado con *algoritmo_pd(**bitmask, índice**, ...)*, donde ***índice*** nos indica la ciudad en la que nos encontramos y ***bitmask*** nos indica las ciudades que ya han sido visitadas. Si el bit la ciudad *i-esima* está en la máscara, significa que esa ciudad *i-ésima* está visitada.

Este algoritmo nos devuelve la mínima distancia para visitar X (número en el conjunto de bits de la máscara) ciudades correspondientes al orden de aparición en la máscara, donde la última ciudad visitada es aquella que aparece en el índice actual.

La relación entre los estados se consigue a través de nuestro estado inicial, que será la que tiene **bitmask** e **índice** a 0. Esto nos indica que actualmente nos situamos en la primera ciudad y que la máscara esté a 0 indica que no se ha visitado ninguna ciudad hasta el momento.

Nuestro estado final será aquel que tiene cualquier **índice** y la máscara sea LIMIT_MASK = (1 << N) – 1, siendo N el número de ciudades.

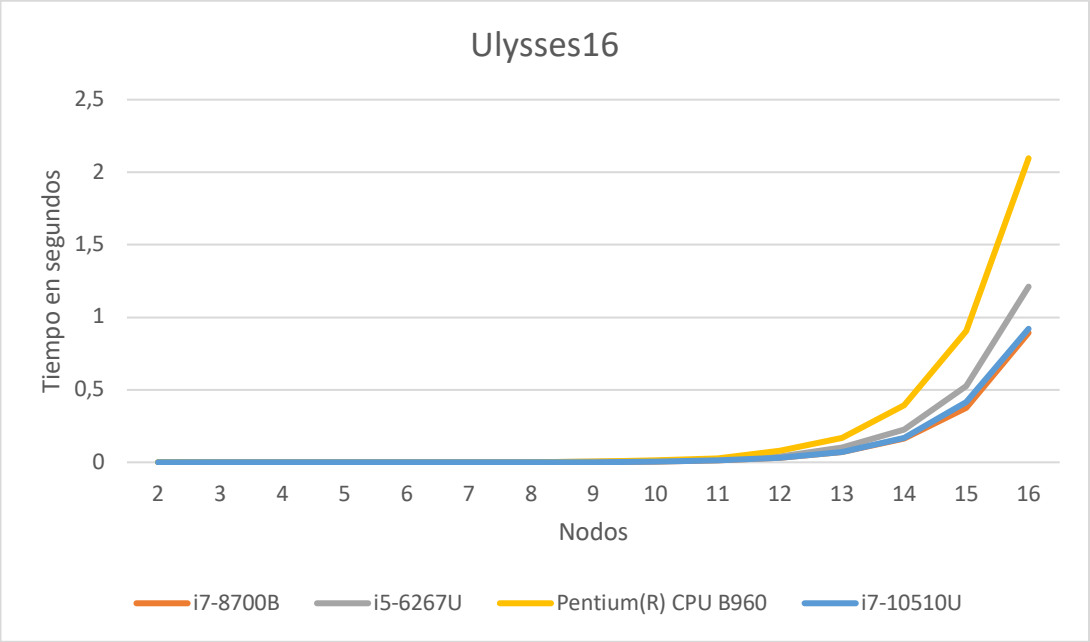
El estado intermedio, será el que calcule la mínima distancia para visitar todas las ciudades estando en una ciudad **índice** *i* y habiendo visitado ya nuevoBitmask es igual al nuevoMin y habiendo visitado (**nuevoBitmask | (1 << actual)**) ciudades.

Por lo tanto, iteramos por todas las posibles ciudades teniendo en cuenta que nuevoBitmask tiene el bit *i-ésimo* a 0, lo que significa que dicha ciudad no ha sido visitada.

Cuando nuestro bitmask sea LIMIT_MASK, significará que hemos visitado todas las ciudades del conjunto. Por ello, debemos añadir la distancia de la última ciudad visitada a la posición inicial.

2.1.2. Tiempos obtenidos

A la hora de la ejecución hemos usado tres de los ejemplos de la práctica anterior, *ulysses16*, *ulysses22* y *att48*. *Ulysses16* posee 16 nodos, por lo que he calculado el tiempo que tarda para calcular con 2 nodos, con 3 nodos con 4 nodos, etc. De esta forma podremos ajustar la curva del tiempo empírico a la curva teórica. De igual modo lo he realizado con *ulysses22* y *att48*.



Esta gráfica nos muestra la ejecución de *ulysses16* en cuatro computadores distintos. Todos ellos tienen como sistema operativo Ubuntu.

Como podemos observar, la función tiende a ser exponencial, por lo que no hemos logrado ejecutar más de 22 ciudades, ya que a partir de este valor el tiempo de ejecución era superior a los 4 minutos.

```
Tiempo (segs) de ejecución del algoritmo dinámico: 0,892883
Coste del camino: 71
Recorrido: 0->2->1->3->7->13->6->5->14->4->10->8->9->11->12->15->0
```

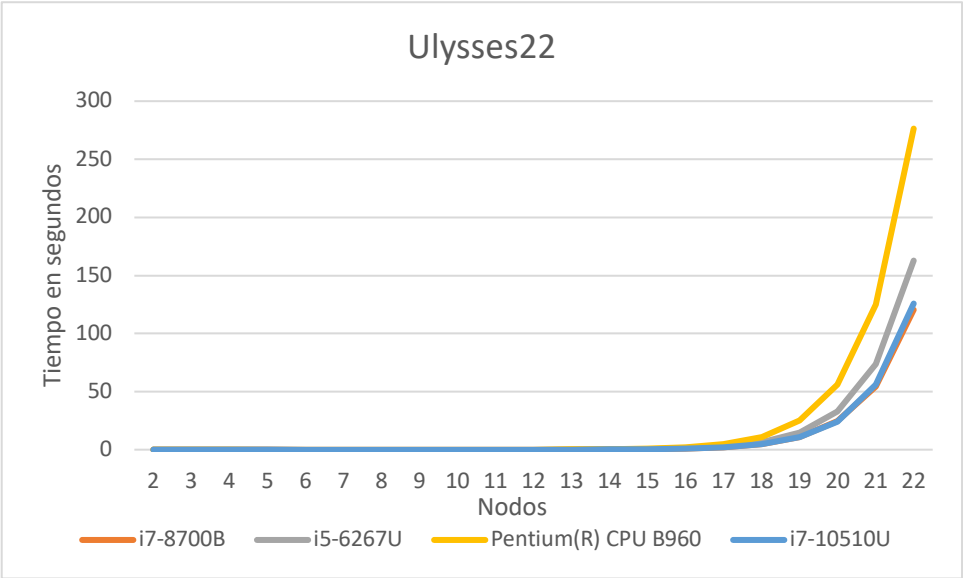
La ejecución del algoritmo nos da el resultado anterior. Si lo comparamos con la ejecución de los algoritmos voraces las soluciones que nos daban eran:

```
Tiempo (segs) de ejecución del algoritmo CERCANÍA: 0,00005
1 -> 8 -> 16 -> 13 -> 14 -> 12 -> 7 -> 6 -> 15 -> 5 -> 9 -> 10 -> 4 -> 2 -> 3 -> 11 -> 1
El coste del camino es: 104
```

```
Tiempo (segs) de ejecución del algoritmo INSERCIÓN: 0,000056
5 -> 15 -> 6 -> 7 -> 13 -> 12 -> 14 -> 8 -> 4 -> 2 -> 3 -> 1 -> 16 -> 10 -> 9 -> 11 -> 5
El coste del camino es: 83
```

```
Tiempo (segs) de ejecución del algoritmo INTERCAMBIO: 0,00005
5 -> 15 -> 6 -> 7 -> 13 -> 12 -> 14 -> 8 -> 4 -> 2 -> 3 -> 1 -> 16 -> 10 -> 9 -> 11 -> 5
El coste del camino es: 83
```

Por lo que, la ejecución con programación dinámica frente a todas las soluciones voraces obtiene una bastante mejor solución para el mismo problema. Sin embargo, el tiempo de ejecución del algoritmo de programación dinámica es ligeramente más elevado que el de los algoritmos voraces.



Si observamos ahora el resultado con *ulysses22* vemos que es una gráfica bastante similar al *ulysses16*. No obstante, debido a que este tiene más nodos vemos un tiempo de ejecución notoriamente más exponencial.

```
Tiempo (segs) de ejecución del algoritmo dinámico: 120,433
Coste del camino: 72
Recorrido: 0->2->1->16->17->3->21->7->11->12->13->6->5->14->4->10->8->9->18->19->20->15->0
```

La ejecución de nuestro algoritmo nos da el anterior resultado. De nuevo, lo comparamos con los algoritmos voraces obtenidos en la anterior práctica.

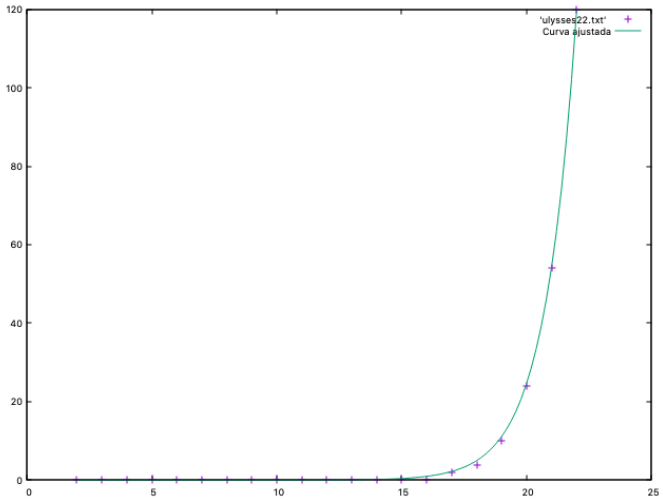
```
Tiempo (segs) de ejecución del algoritmo CERCANÍA: 0,000014
1 -> 8 -> 16 -> 22 -> 4 -> 17 -> 2 -> 3 -> 18 -> 12 -> 14 -> 13 -> 7 -> 6 -> 15 -> 5 -> 9 -> 10 -> 19 -> 20 -> 11 -> 21 -> 1
El coste del camino es: 98
```

```
Tiempo (segs) de ejecución del algoritmo INSERCIÓN: 0,000033
5 -> 6 -> 7 -> 11 -> 9 -> 10 -> 19 -> 20 -> 21 -> 13 -> 12 -> 16 -> 1 -> 3 -> 2 -> 17 -> 18 -> 4 -> 22 -> 8 -> 14 -> 15 -> 5
El coste del camino es: 87
```

```
Tiempo (segs) de ejecución del algoritmo INTERCAMBIO: 0,00015
7 -> 6 -> 5 -> 11 -> 9 -> 10 -> 19 -> 20 -> 21 -> 12 -> 13 -> 16 -> 1 -> 3 -> 2 -> 17 -> 18 -> 4 -> 22 -> 8 -> 14 -> 15 -> 7
El coste del camino es: 79
```

He de comentar que esta vez, el algoritmo que usa programación dinámica no sólo vuelve a tardar más en comparación con los algoritmos voraces, si no que se trata de un 8.028% de aumento respecto al de los voraces, siendo esta una cifra muy alta. Sin embargo, nuevamente el algoritmo que usa programación dinámica vuelve a obtener el mejor resultado en cuanto a coste del camino y, por tanto, el que mejor solución obtiene.

A continuación, expongo los puntos obtenidos con la ejecución de *ulysses22* ajustada a la curva teórica con el uso de *gnuplot*. De este modo obtendré la ecuación empírica de la eficiencia para el computador con *i7-8700B*.



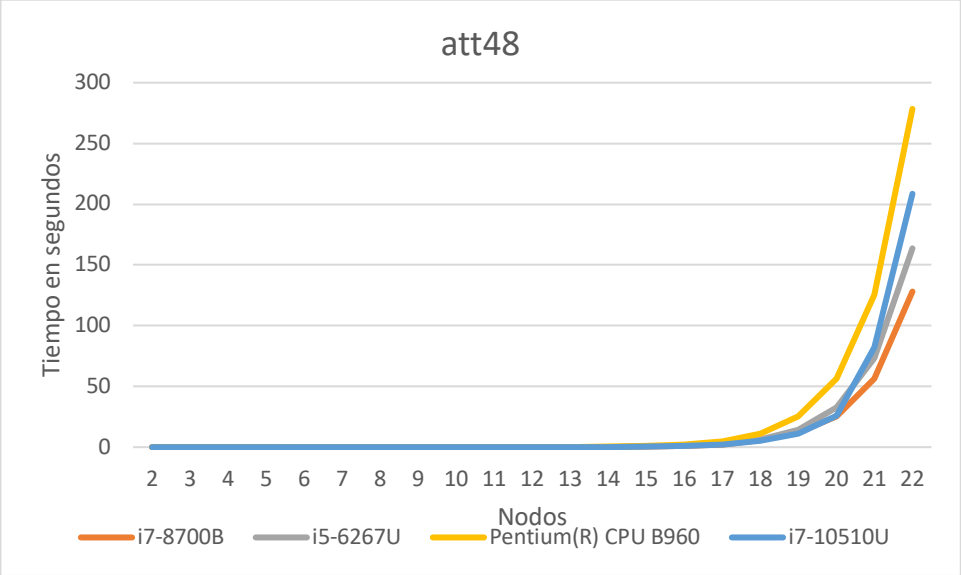
La función que define la curva para el procesador *i7-8700B* en *ulysses22* por lo tanto sería:

$$f(x) = 0{,}000102726 \cdot x^2 \cdot 0{,}000573007 \cdot 2^x$$

El error que obtenemos es el siguiente:

```
Asymptotic Standard Error
=========================
+/- 1.608e+07     (1.565e+13%)
+/- 8.97e+07      (1.565e+13%)
```

A través de esta ecuación podemos obtener para cualquier número de nodos x su tiempo de ejecución *f(x)*.



En la gráfica anterior observamos nuevamente cuatro computadores, esta vez ejecutando el caso *att48*. Este, sin embargo, ha sido ejecutado hasta 22 ciudades nuevamente, ya que tras el estudio de la función empírica caso anterior, *ulysses22*, hemos visto que con un número de vértices superior a 22 se tarda demasiado tiempo en ejecutar.

```
Tiempo (segs) de ejecución del algoritmo dinámico: 128.052
Coste del camino: 24154
Recorrido:   0->7->8->14->17->6->5->18->16->19->11->10->12->20->9->3->1->4->13->2->21->15->0
```

La ejecución de nuestro algoritmo nos da el resultado anterior. Nuevamente, lo comparamos con los algoritmos voraces obtenidos en la anterior práctica.

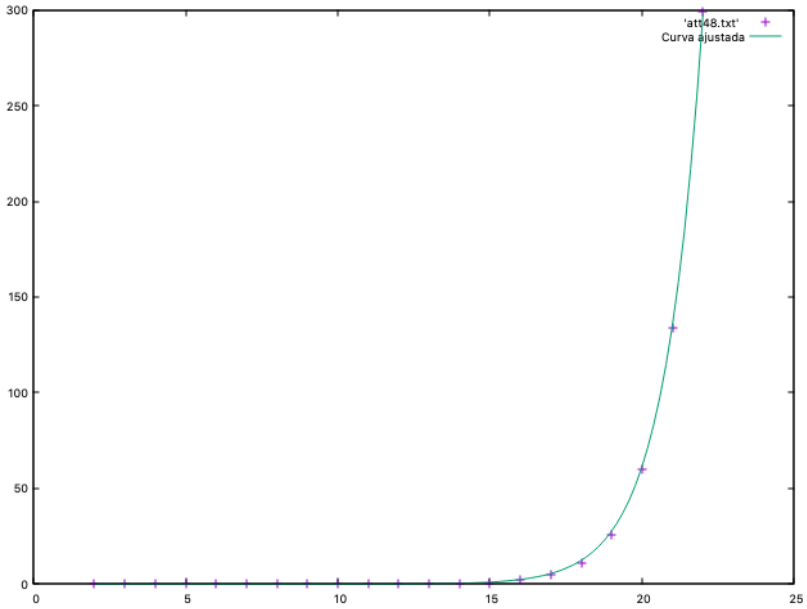
```
Tiempo (segs) de ejecución del algoritmo CERCANÍA: 0,000027
1 -> 9 -> 8 -> 22 -> 16 -> 3 -> 14 -> 13 -> 21 -> 11 -> 12 -> 15 -> 20 -> 18 -> 7 -> 6 -> 19 -> 17 -> 5 -> 2 -> 4 -> 10 -> 1
El coste del camino es: 29346
```

```
Tiempo (segs) de ejecución del algoritmo INSERCIÓN: 0,000081
4 -> 10 -> 5 -> 14 -> 13 -> 21 -> 11 -> 12 -> 20 -> 17 -> 19 -> 6 -> 7 -> 18 -> 15 -> 9 -> 8 -> 1 -> 16 -> 22 -> 3 -> 2 -> 4
El coste del camino es: 30678
```

```
Tiempo (segs) de ejecución del algoritmo INTERCAMBIO: 0,000963
2 -> 4 -> 10 -> 5 -> 21 -> 13 -> 11 -> 12 -> 20 -> 17 -> 19 -> 6 -> 7 -> 18 -> 15 -> 9 -> 8 -> 1 -> 16 -> 22 -> 3 -> 14 -> 2
El coste del camino es: 29198
```

Observando los resultados, vemos que existe un símil entre todas las comparaciones. Una vez más el algoritmo que usa programación dinámica vuelve a tardar mucho más que los voraces (un 15.808% más), pero este nos da el menor de los costes.

De igual modo obtenemos con *gnuplot* la ecuación empírica de la eficiencia. Ajustamos nuestros puntos a la curva teórica.



La función que define la curva para el procesador *i7-8700B* en *att48* por lo tanto sería:

$$f(x) = 0{,}000228646 \cdot x^2 \cdot 0{,}000641378 \cdot 2^x$$

El error que obtenemos es el siguiente:

```
Asymptotic Standard Error
=========================
+/- 7.539e+06    (3.297e+12%)
+/- 2.115e+07    (3.297e+12%)
```

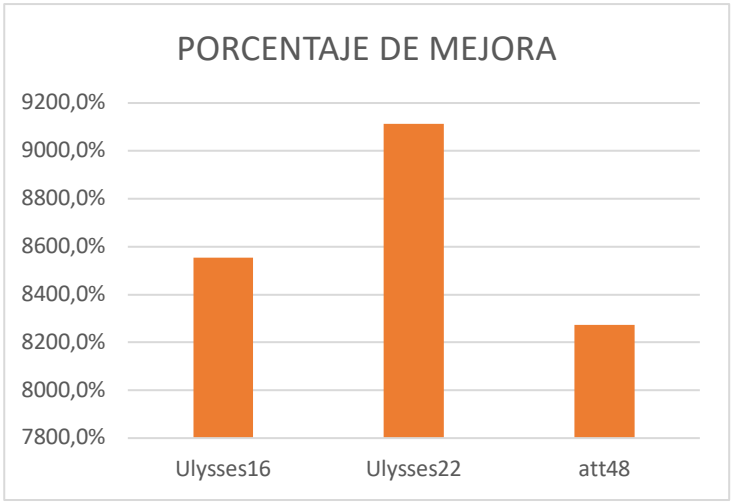
A través de esta ecuación podemos obtener para cualquier número de nodos x su tiempo de ejecución *f(x)* y notamos que es prácticamente igual que la de *ulysses22*.

Por último, he de añadir que no ejecutamos el caso *a280* debido a que posee 280 vértices. Esta cantidad de vértices es tan elevada que según podemos calcular con la función de eficiencia empírica, tardaríamos 46122882673553200 siglos en computarlo con mi procesador. Este ejemplo, por tanto, es inviable para calcularlo con este algoritmo.

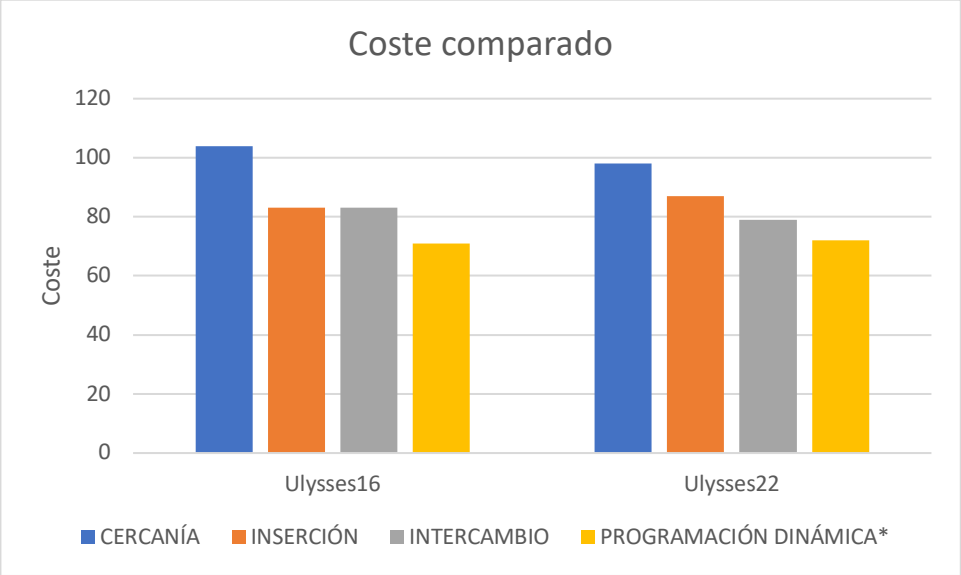
2.1.3. Tabla comparativa con tiempos de ejecución

TIEMPO	CERCANÍA	INSERCIÓN	INTERCAMBIO	PROGRAMACIÓN DINÁMICA	PORCENTAJE TIEMPOS
Ulysses16	0,00005	0,000056	0,00005	0,892883	178576600%
Ulysses22	0,000014	0,000033	0,00015	120,433	86023571428,6%
att48*	0,000027	0,000081	0,000963	128,052	47426666666,7%
a280	0,000449	0,001413	0,224771		
*= hasta 22 nodos en i7-8700B					

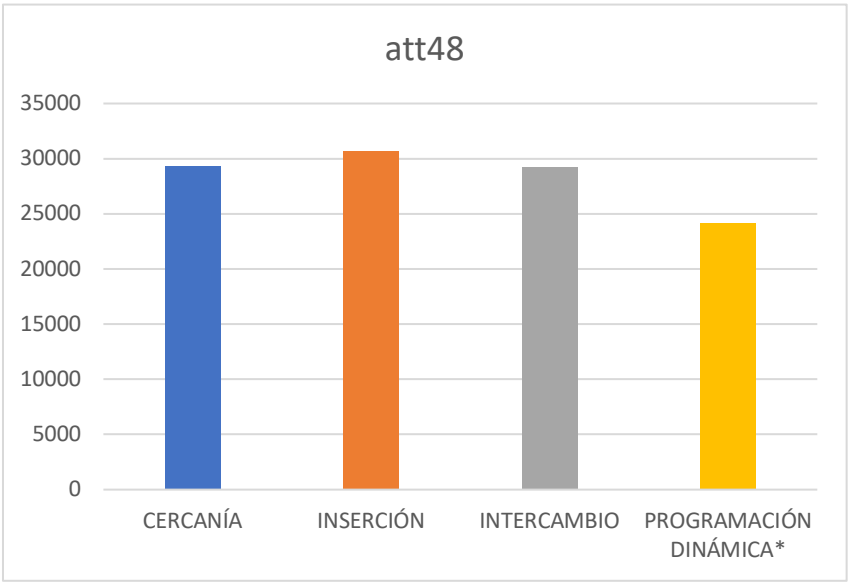
COSTE	CERCANÍA	INSERCIÓN	INTERCAMBIO	PROGRAMACIÓN DINÁMICA*	PORCENTAJE DE MEJORA
Ulysses16	104	83	83	71	8554,2%
Ulysses22	98	87	79	72	9113,9%
att48*	29346	30678	29198	24154	8272,5%
a280					
*= hasta 22 nodos en i7-8700B					



En la gráfica podemos observar cómo nuestro algoritmo en programación dinámica mejora la obtención del resultado, llegando, en su mejor caso, a mejorar el *Ulysses22* un 9114% respecto a la mejor solución que encontramos mediante el uso de algoritmos voraces.

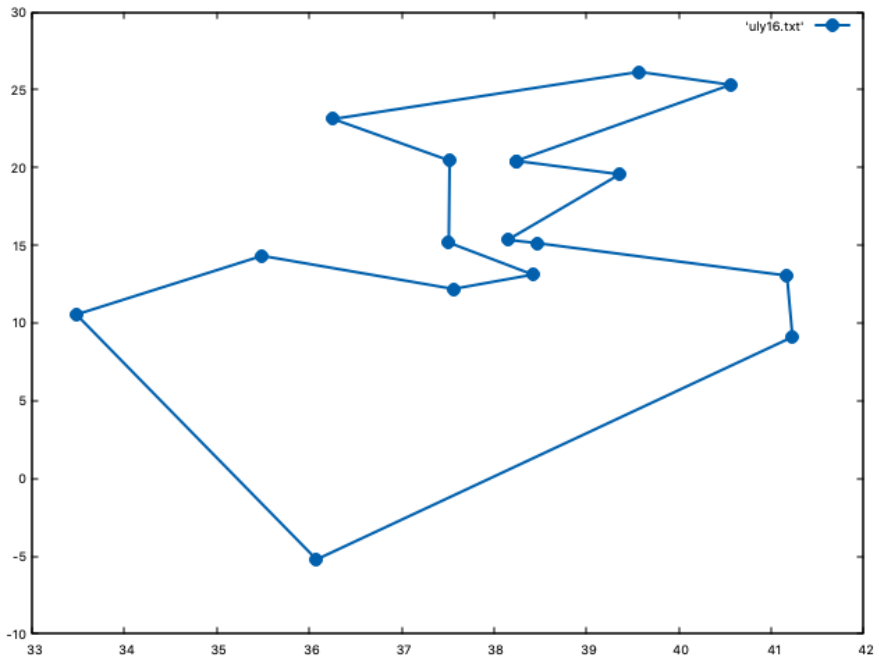


Comparando los costes obtenidos con *Ulysses*, los cuales tienen ambos datos relacionados, obtenemos la gráfica anterior, en el que observamos lo dicho en el apartado 2.1.2. de forma visual. Aquel que tiene menor coste es el algoritmo de programación dinámica y, por tanto, el mejor.

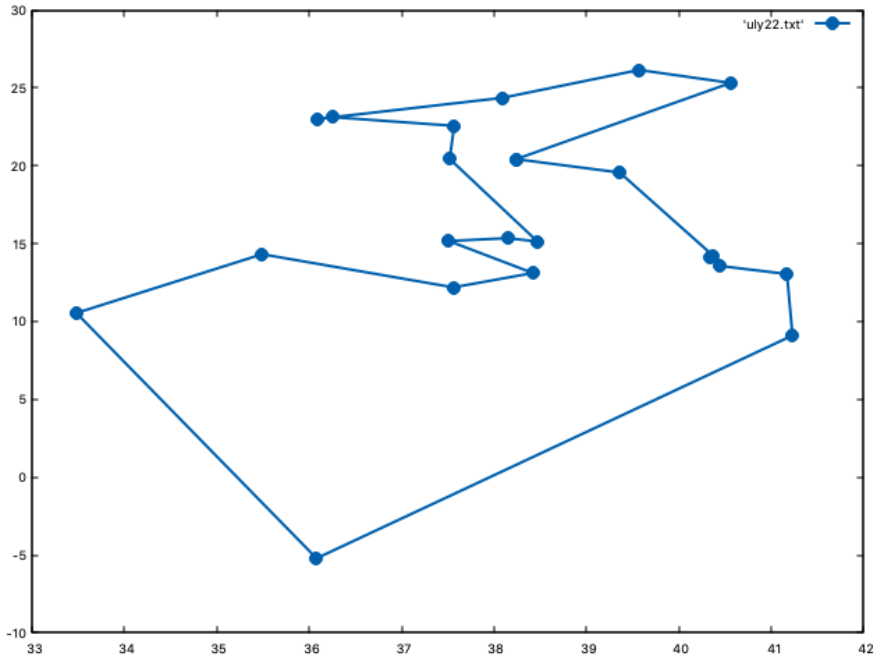


De igual forma, en la gráfica anterior observamos el coste obtenido con los datos de *att48*, en el que vemos que nuevamente programación dinámica es el que menor coste obtiene.

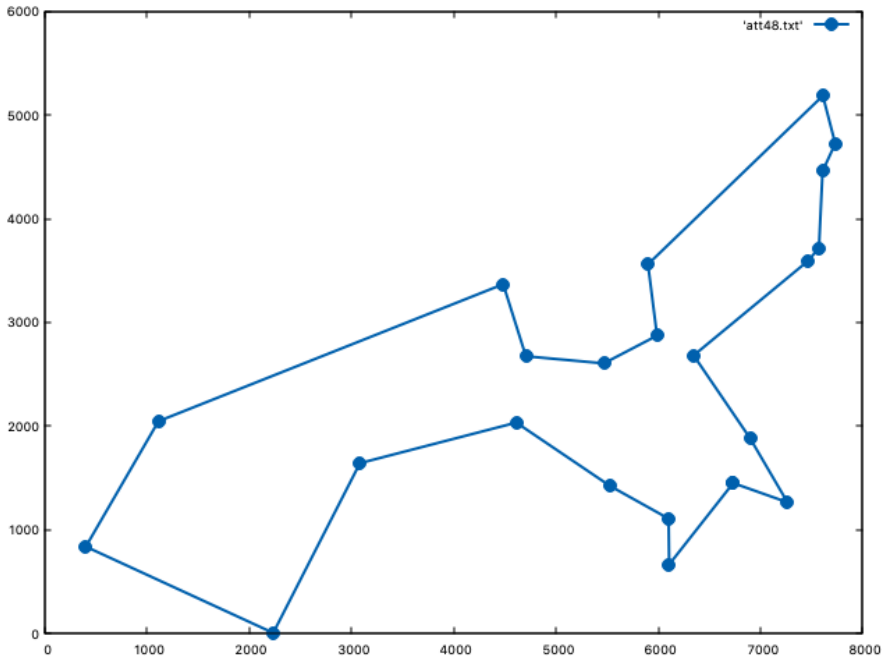
2.1.4. Representación de los recorridos obtenidos



1: Ulysses16



2: Ulysses22



3: att48 hasta el vértice 22

Capítulo 3: Conclusiones

Finalmente, tras haber diseñado el algoritmo que resuelve el problema del viajante de comercio usando programación dinámica, con su correspondiente pseudocódigo, y habiendo obtenido los tiempos de ejecución en 4 computadoras que poseen especificaciones diferentes, y habiendo comparado dichos tiempos con los que obtuvimos en la práctica anterior a través de los algoritmos voraces, concluimos que el uso de programación dinámica para abordar este tipo de problemas es el más adecuado cuando se puede encontrar, como en este caso, una forma recurrente de resolver un gran problema a través de la subdivisión del mismo en partes más pequeñas.

Además, he comprobado que el uso de este algoritmo es bueno para problemas cuyo tamaño no sea muy grande debido a su orden de eficiencia. En este caso, una vez que superamos las 22 ciudades el tiempo de ejecución tiende a infinito, motivo por el que uno de los 4 casos propuestos no ha podido ser ejecutado (*a280*). En relación con el tiempo, es importante decir también que, en comparación con los algoritmos voraces, tarda bastante más, sobre todo cuando el problema se hace cada vez más grande. En relación con el coste obtenido, la solución del algoritmo que usa programación dinámica es mucho mejor que el de los algoritmos voraces.