Universidad De Granada

Práctica 5

Algoritmo de vuelta atrás
Y
Ramificación y Poda

Víctor José Rubia López
B3
Fecha de entrega 8/06/2020

Contenido

Capítulo 1: Cena de gala

1.1. Introducción

El objetivo que hemos de perseguir con este problema es diseñar un algoritmo *Backtracking* que lo resuelva. Además, realizaremos un estudio empírico de su eficiencia.

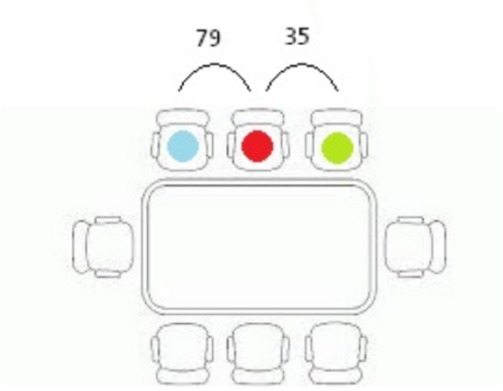
Backtracking es una técnica algorítmica general que considera cada combinación posible que se pueda encontrar, para resolver un problema computacional. Recursivamente, intentará construir una solución incrementalmente, parte por parte, eliminando aquellas soluciones que no satisfacen las restricciones del problema en cualquier nivel del árbol de búsqueda.

El problema consiste en encontrar la mayor conveniencia entre comensales de una cena de gala. El número de invitados será n, que se sentarán alrededor de una gran mesa rectangular, de forma que cada invitado tiene a dos comensales sentados a su lado. El nivel total de conveniencia es el propio de cada invitado sumado al nivel de conveniencia de los dos comensales que tiene a su lado.

Veremos como con la técnica *Backtracking*, podemos conseguir la solución óptima.

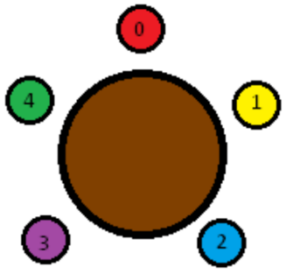
1.2. Análisis del problema

En nuestro problema tenemos n invitados. Cada invitado tiene una afinidad para cada uno de los demás invitados. Esto lo representaremos con un número entre 0 y 100. La conveniencia total es la suma de las conveniencias de cada invitado con los que tiene al lado. Buscamos la secuencia de invitados con la que conseguimos mayor nivel de conveniencia.



En esta imagen representamos la afinidad que tiene el invitado rojo con el azul (79) y el verde (35). Con los demás invitados también tendrá x afinidad.

Un ejemplo de invitados a la cena de gala sería la siguiente:



Afinidad:
0-1 (30) 1-2 (80) 2-3 (40) 3-4 (1)
0-2 (50) 1-3 (10) 2-4 (20)
0-3 (60) 1-4 (5)
0-4 (70)

Tenemos que tomar una serie de decisiones, las cuales nos llevarán a un nuevo conjunto de decisiones, donde alguna secuencia de ellas será la solución a nuestro problema.

Para calcularla usaremos *Backtracking*. La eficiencia sería de $O(q(n)n!)$, siendo q un polinomio en n, ya que tenemos que explorar $n!$ nodos.

1.3. Elementos para solucionar el problema

Dada una matriz M[i][j], la matriz de conveniencia entre el comensal i y el comensal j será:

$$\begin{pmatrix} 0 & 15 & 25 \\ 15 & 0 & 10 \\ 25 & 10 & 0 \end{pmatrix}$$

Esta matriz siempre será simétrica y los valores de conveniencia serán asignados aleatoriamente, teniendo la diagonal de 0s.

La solución será representada en un vector que tiene longitud N (nº de invitados) y que guarde en cada posición el valor del invitado que se sienta en la posición i. Su restricción debe ser que tome un valor entero de 1 a N (nº de invitados).

1.4. Explicación del Pseudocódigo

Para comenzar, sentamos al comensal para el que estamos calculando la posible solución. Recorremos todos los comensales y, si el comensal que estamos mirando no está sentado, pasamos a evaluar, calculando la cota que tenemos hasta el momento, y descendemos en profundidad para evaluar sus hijos. Si estamos en un nodo hoja, entonces calculamos la cota total que obtenemos con todos los comensales sentados y, si la cota que obtenemos es mejor que la máxima que hemos obtenido hasta el momento, la actualizamos y guardamos la solución. Si no estamos en un nodo hoja seguimos descendiendo. Por último, levantamos al comensal para seguir buscando si hay alguna opción mejor.

1.5. Pseudocódigo

```
función backtracking(sol, sol_parcial, sentados, comensal_actual, nivel)
variables  Matriz,  sol_final[N],  sol_parcial[N],  sentados[N]:=false,
comensal_actual, nivel, val_max:=0

Sentados[comensal_actual]:=true
sol_parcial[nivel-1]:=comensal_actual
para todo i en N hacer
   si sentados[i]==false entonces
      valor_actual=CalcularSolucionActual(sol_parcial)
      backtracking(sol, sol_parcial, sentados, i, nivel+1)
      si nodo_actual==nodo_hoja entonces
         valor_actual:=CalcularSolucionActual()
         si valor_actual es mayor que valor_maximo entonces
            sol_final:=sol_actual
            valor_maximo:=valor_actual
         fsi
      sino
         valor_actual = CalcularSolucionActual(sol_parcial)
      fsino
      fsi
      Sentados[i]:=false
   fsi
fpara
```

1.6. Casos de ejecución

$$\begin{pmatrix} 0 & 17 & 97 & 20 & 65 & 36 & 32 \\ 17 & 0 & 21 & 2 & 79 & 53 & 86 \\ 97 & 21 & 0 & 63 & 15 & 20 & 94 \\ 20 & 2 & 63 & 0 & 71 & 49 & 91 \\ 65 & 79 & 15 & 71 & 0 & 54 & 25 \\ 36 & 53 & 20 & 49 & 54 & 0 & 60 \\ 32 & 86 & 94 & 91 & 25 & 60 & 0 \end{pmatrix}$$

```
Para 9 comensales se tarda en ordenar 0.309503
Se sientan así:
0 2 8 3 1 7 6 4 5
(Donde el primero se sienta al lado del último)
Número de nodos explorados:109600
```

$$\begin{pmatrix} 0 & 52 & 42 & 97 & 14 & 45 & 45 & 41 & 51 & 53 & 63 \\ 52 & 0 & 92 & 47 & 87 & 83 & 82 & 45 & 96 & 36 & 60 \\ 42 & 92 & 0 & 85 & 81 & 21 & 16 & 80 & 57 & 68 & 68 \\ 97 & 47 & 85 & 0 & 75 & 94 & 50 & 64 & 66 & 19 & 42 \\ 14 & 87 & 81 & 75 & 0 & 96 & 3 & 86 & 44 & 42 & 46 \\ 45 & 83 & 21 & 94 & 96 & 0 & 95 & 50 & 44 & 35 & 22 \\ 45 & 82 & 16 & 50 & 3 & 95 & 0 & 99 & 86 & 23 & 45 \\ 41 & 45 & 80 & 64 & 86 & 50 & 99 & 0 & 97 & 88 & 9 \\ 51 & 96 & 57 & 66 & 44 & 44 & 86 & 97 & 0 & 35 & 82 \\ 53 & 36 & 68 & 19 & 42 & 35 & 23 & 88 & 35 & 0 & 48 \\ 63 & 60 & 68 & 42 & 46 & 22 & 45 & 9 & 82 & 48 & 0 \end{pmatrix}$$

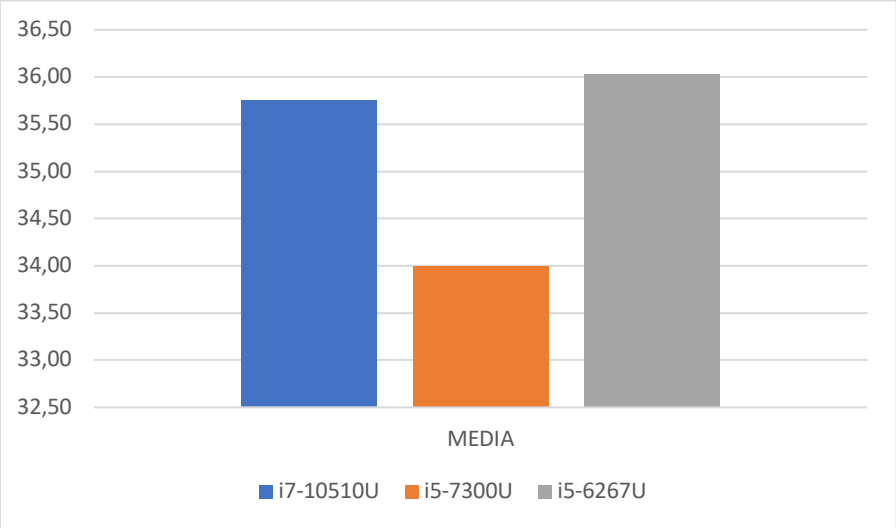
```
Para 11 comensales se tarda en ordenar 30.072
Se sientan así:
0 3 9 10 4 5 6 7 2 8 1
(Donde el primero se sienta al lado del último)
Número de nodos explorados:9864100
```

$$\begin{pmatrix} 0 & 95 & 14 & 35 & 22 & 93 & 56 & 3 & 22 & 82 & 89 & 59 \\ 95 & 0 & 57 & 68 & 94 & 51 & 70 & 68 & 57 & 74 & 32 & 41 \\ 14 & 57 & 0 & 51 & 94 & 58 & 70 & 63 & 31 & 93 & 38 & 67 \\ 35 & 68 & 51 & 0 & 2 & 54 & 43 & 31 & 94 & 21 & 100 & 6 \\ 22 & 94 & 94 & 2 & 0 & 10 & 29 & 14 & 22 & 9 & 60 & 83 \\ 93 & 51 & 58 & 54 & 10 & 0 & 63 & 1 & 61 & 40 & 3 & 98 \\ 56 & 70 & 70 & 43 & 29 & 63 & 0 & 78 & 63 & 80 & 34 & 27 \\ 3 & 68 & 63 & 31 & 14 & 1 & 78 & 0 & 61 & 77 & 23 & 40 \\ 22 & 57 & 31 & 94 & 22 & 61 & 63 & 61 & 0 & 94 & 23 & 72 \\ 82 & 74 & 93 & 21 & 9 & 40 & 80 & 77 & 94 & 0 & 51 & 80 \\ 89 & 32 & 38 & 100 & 60 & 3 & 34 & 23 & 23 & 51 & 0 & 48 \\ 59 & 41 & 67 & 6 & 83 & 98 & 27 & 40 & 72 & 80 & 48 & 0 \end{pmatrix}$$

```
Para 12 comensales se tarda en ordenar 443.126
Se sientan así:
0 4 7 10 3 1 6 5 9 8 11 2
(Donde el primero se sienta al lado del último)
Número de nodos explorados:108505111
```

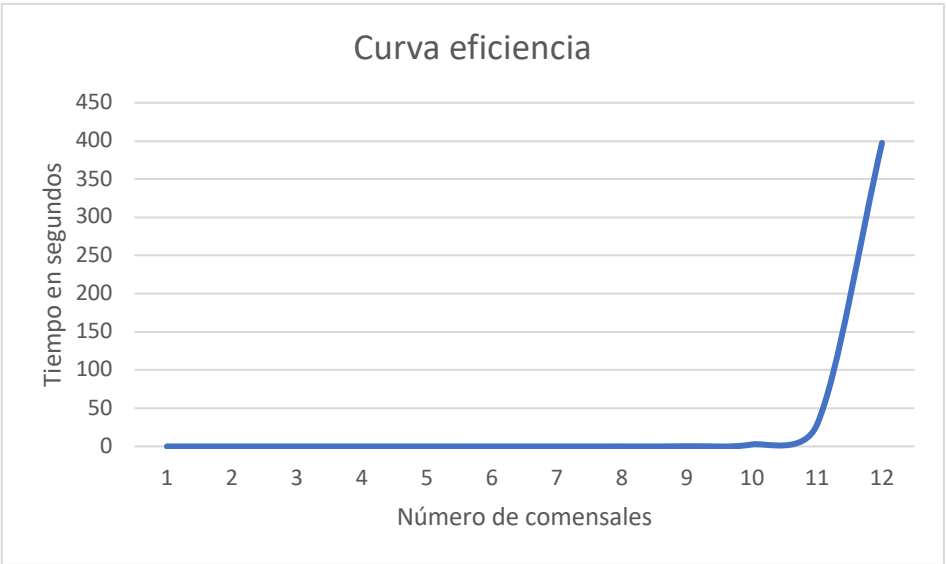
1.7. Eficiencia empírica

La siguiente gráfica es una representación de la media que ha tardado en ejecutarse el programa para tres procesadores distintos para los casos de tener de 2 a 12 comensales. Vemos cómo en el procesador (i5-7300U), obtenemos que sus tiempos son de media inferiores que los demás.

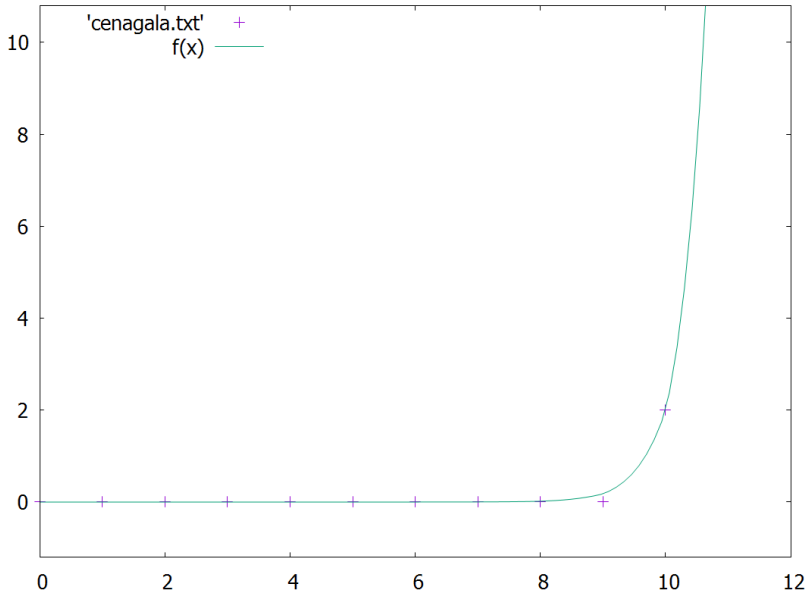


CENA DE GALA

TIEMPO EN SEGUNDOS	
TAM, MATRIZ	i7-10510U
1	0,0000005
2	0,0000042
3	0,0000077
4	0,0000276
5	0,0000986
6	0,000538
7	0,004318
8	0,0260612
9	0,2686
10	2,57094
11	28,5846
12	397,509
MEDIA	35,75



1.8. Eficiencia híbrida



Hemos usado *gnuplot* para obtener la eficiencia híbrida. Sabiendo la eficiencia teórica y, habiendo obtenido los resultados de tiempo empíricos, procedemos a ajustar los puntos a la ecuación $f(x) = a \cdot x \cdot x!$, siendo $a{\cdot}x$ el polinomio *q(n)*.

Obtenemos la siguiente función de eficiencia híbrida, que se ajusta a los puntos obtenidos:

$$f(x) = 0.0000000546675 \cdot x \cdot x!$$

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a                = 5.46675e-08     +/- 1.564e-09    (2.86%)
```

Capítulo 2: Problema del Viajante de Comercio

2.1. Introducción

Nuestro objetivo será resolver el problema del viajante de comercio utilizando para ello un enfoque Branch and Bound (Ramificación y Poda) y otro usando Backtracking (Vuelta atrás). Compararemos la eficiencia entre ambos algoritmos.

Como sabemos, el problema del viajante de comercio es de los denominados problemas NP-Completos. Esto quiere decir que no se puede encontrar una solución óptima mediante algoritmos Greedy. Sin embargo, a través de Branch and Bound encontraremos la solución óptima.

2.2. Diseño del algoritmo Branch and Bound

El problema del viajante de comercio ya se ha comentado y utilizado en las dos prácticas anteriores, donde se estudiaron los algoritmos voraces para encontrar soluciones razonables y no óptimas necesariamente. Si queremos encontrar una solución óptima es necesario usar métodos más potentes y costosos, como la programación dinámica, la vuelta atrás y la ramificación y poda, que exploren el espacio de posibles soluciones de forma más exhaustiva.

Así, un algoritmo de vuelta atrás comenzaría en la ciudad 0 (qué al tratarse de un recorrido, la ciudad de inicio y fin es la misma) e intentaría incluir como parte del recorrido la siguiente ciudad aun no visitada, continuando de este modo hasta completar un recorrido.

Para agilizar la búsqueda de la solución se deben considerar como ciudades válidas para una posición únicamente aquellas que satisfagan las restricciones del problema. En nuestro caso esas serán las ciudades que aún no hemos visitado. Cuando para un nivel no queden más ciudades válidas, el algoritmo hace una vuelta atrás proponiendo una nueva ciudad válida para el nivel anterior.

Para usar un algoritmo de Ramificación y Poda, es necesario utilizar una cota local. Esta cota local es un valor menor o igual al verdadero coste de la mejor solución que podremos obtener a partir de la solución parcial en la que nos encontremos.

Una posible alternativa es, como conocemos cuáles son las ciudades que aún faltan por visitar, estimar de forma optimista el coste que todavía nos queda, que será, para cada ciudad, el coste del mejor, que será el menor de los arcos salientes de esa ciudad. La suma de los costes de esos arcos sumado al coste del camino ya acumulado es una cota local en el sentido que hemos descrito anteriormente.

Como criterio para seleccionar el siguiente nodo que hay que expandir del árbol de búsqueda, se empleará el criterio LC o "más prometedor". En este caso consideraremos como nodo más prometedor aquel que presente el menor valor de cota local. Para ello se debe utilizar una cola con prioridad que almacene los nodos ya generados (nodos vivos).

Además de devolver el costo de la solución encontrada junto con el tour correspondiente, se deben de obtener también resultados relativos a la complejidad del cálculo como el número de nodos expandidos, tamaño máximo de la cola con prioridad de nodos vivos, números de veces que se realiza la poda y el tiempo empleado para resolver el problema.

Las pruebas del algoritmo las realizaremos con los mismos datos empleados en la práctica 4, ya que el tamaño de problemas que podemos abordar con esta técnica es mucho más reducido que los métodos voraces.

2.2.1. Pseudocódigo

```
Funcion tspbb(matriz_costes, vector_ciudades, vector_dist_min)
variables matriz_costes, vector_ciudades[N], vector_dist_min, cola, sol_final
Nodo n.generarnodosvivos(vector_ciudades[0])
mientras !cola.vacia() hacer
   nodo:=cola.top()
   si EsHoja(nodo) and nodo.cotalocal es mayor que cota_global entonces
      sol_final:= nodo.solucion
      cota_global:=nodo.cotalocal
   fsi
   si nodo.cotalocal es menor que cota_global entonces
      cola.add(nodo.generarhijos())
      cola.push()
   sino
      devuelve solucion_final
   fsino
   fsi
fmientras
```

2.2.2. Explicación del algoritmo

Para usar *Branch and Bound* necesitaremos crearnos una estructura de datos la cual llamaremos *Nodo.* Esto nos servirá para manejar árboles.

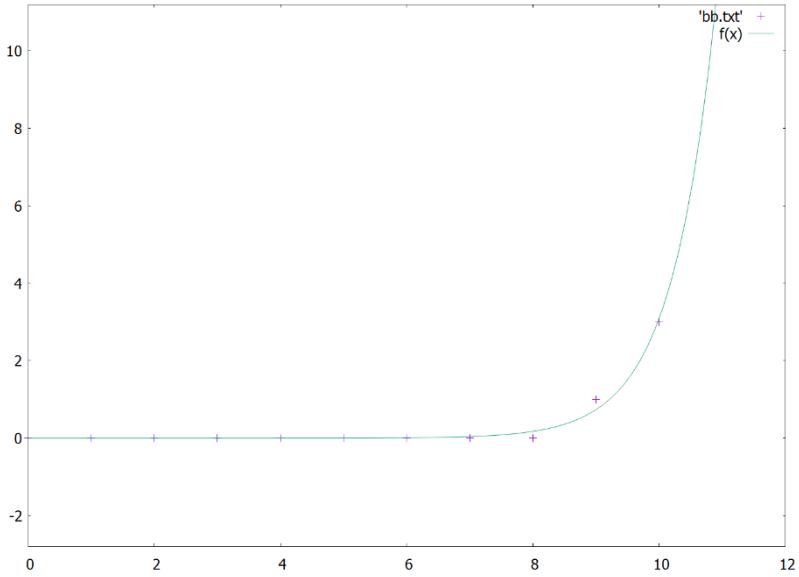
Usaremos una cola con prioridad para almacenar los nodos vivos, tal y como se indica en el guion de prácticas. Además, tendremos un vector de booleanos que nos indique qué ciudades han sido visitadas hasta el momento.

Comenzaremos el algoritmo generando el nodo inicial que será, para comenzar, la ciudad 0. Tras introducirla en la cola con prioridad, realizaremos un bucle mientras haya nodos vivos. En dicho bucle, obtendremos el primer nodo de la cola con prioridad y, si es un nodo hoja cuya cota local sea menor que la global, actualizaremos la cota global y la solución final. Además, si el nodo es capaz de

alcanzar una mejor solución, calcularemos los nodos vivos. De lo contrario, si no hay ningún nodo favorable en la lista de nodos vivos, devolveremos la mejor solución que será la mejor encontrada hasta entonces.

2.2.3. Eficiencia teórica e híbrida

La eficiencia teórica del algoritmo usando el método Ramificación y Poda es de $O(2^n)$. Por ello, usando los resultados de ejecución obtenidos en el apartado 2.4., obtenemos, a través de *gnuplot*, la eficiencia híbrida.



$$f(x) = 0.00000123748 \cdot 2^{2{,}07027x+0{,}550565}$$

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a                = 1.23748e-06     +/- 6.53e+05     (5.277e+13%)
b                  = 2.07027       +/- 0.218          (10.53%)
c                = 0.550565        +/- 7.612e+11    (1.383e+14%)
```

2.3. Diseño del algoritmo Backtracking

La solución que planteo es muy similar a la aplicada en la cena de gala. Mientras vamos recorriendo el árbol en profundidad, calcularemos cuánto vale en cada nodo la cota local. Podaremos ese nodo (y a todos sus hijos) cuando sea mayor que la global.

2.3.1. Pseudocódigo

```
función tspbacktracking(sol, sol_parcial, ciudades, ciudad_actual, nivel)
variables matriz, sol_final[N], sol_parcial[N], ciudades[N]:=false, ciudad_actual, nivel, val_max:=0
ciudades[ciudad_actual]=true
sol_parcial[nivel-1]=ciudad_actual
para i hasta N hacer
    si ciudades[i] es igual que false entonces
        valor_acutal:=calcularSolucionActual(sol_parcial)
        si cotalocal es menor que cotaglobal entonces
            tspbacktracking(sol, sol_parcial, ciudades, i, nivel+1)
        fsi
        si nodo_actual es igual que nodo_hoja entonces
            valor_actual:=CalcularSolucionActual(sol_parcial)
            si valor_actual es mayor que valormaximo entonces
                sol_final:=sol_actual
                valor_maximo:=valor_actual
            fsi
        fsi
        ciudades[i]:=false
    fsi
fpara
```

2.3.2. Explicación del algoritmo

Al iniciar el algoritmo recurrente, recorreremos todas las ciudades. Si la ciudad en la que nos encontramos en un momento i no ha sido visitada todavía, establecemos su cota local y comprobamos si esta es menor que la cota global. Si esta condición se cumple, bajaremos un nivel en el árbol.

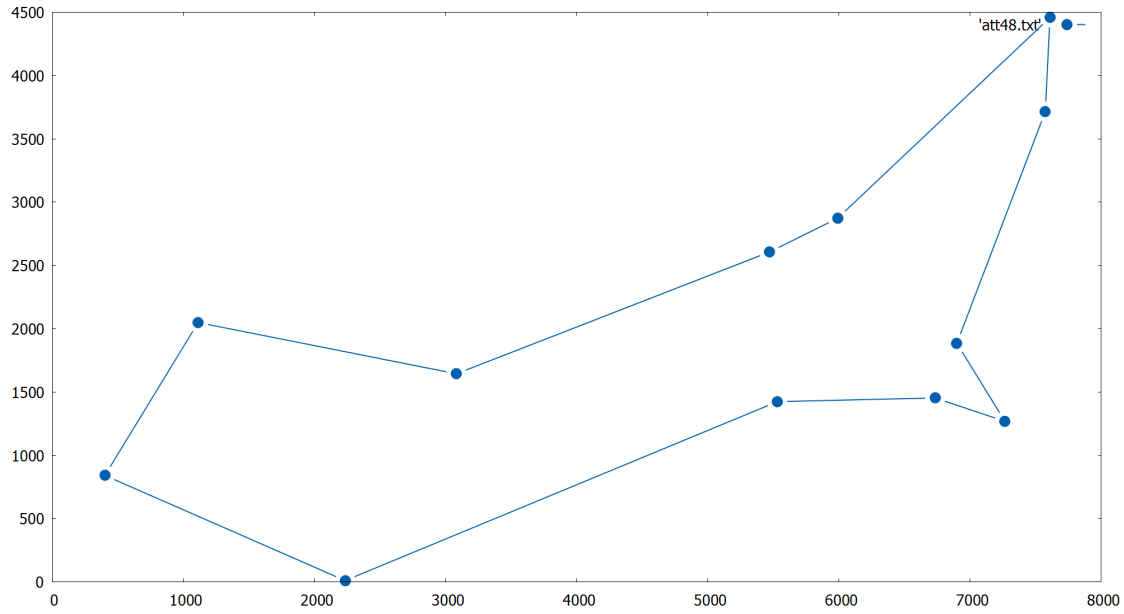
De otra forma, si en el momento i nos encontramos en un nodo hoja, cerraremos el circuito, sumando el coste de la última ciudad con la primera. Además, si la cota que obtenemos es mejor que la que llevamos hasta el momento i la actualizamos.

En el caso de que no estemos en un nodo hoja, deshacemos los cambios que hemos hecho al establecer la cota local.

2.4. Casos de ejecución

A continuación, expongo 4 casos de ejecución tomados de la práctica número 3. Estos casos son *att48, a280, ulysses16* y *ulysses22*. Cabe destacar que, como hemos mencionado anteriormente, durante todo nuestro análisis, hemos observado que la ejecución de este algoritmo tiene un orden de eficiencia muy alta, por lo que solo hemos ejecutado hasta 12 ciudades.

2.4.1. att48



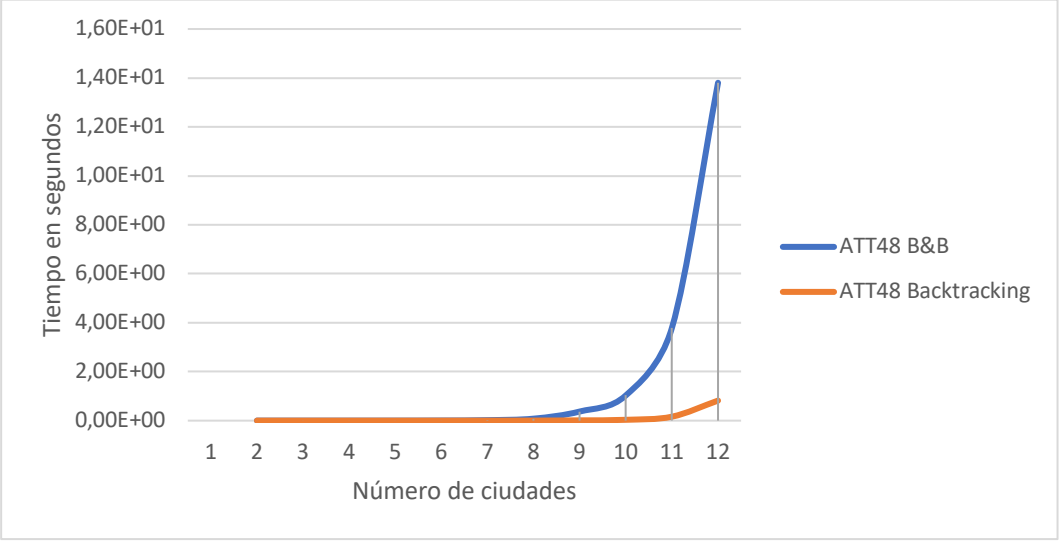
1: Recorrido generado por el algoritmo

DISTANCIA: 19614.6
TOUR: 1 3 2 4 10 5 11 12 6 7 9 8 1
NODOS TOTALES: 479001600
NODOS PODADOS: 478081610
NODOS EXPLORADOS: 919990
TIEMPO (seg): 0.819056

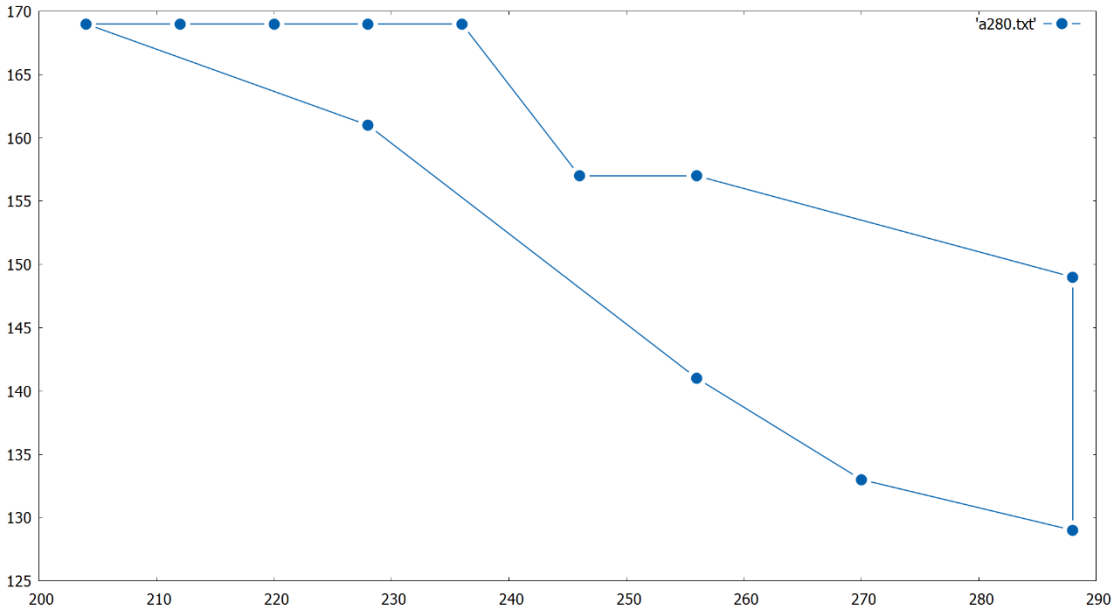
2. Backtracking

DISTANCIA: 19614.6
TOUR: 1 3 2 4 10 5 11 12 6 7 9 8 1
NODOS TOTALES: 479001600
NODOS EXPLORADOS: 120931
NODOS EXPANDIDOS: 442282
NODOS PODADOS: 478880669
TIEMPO (seg): 13.8036

3. Branch & Bound



2.4.2. a280



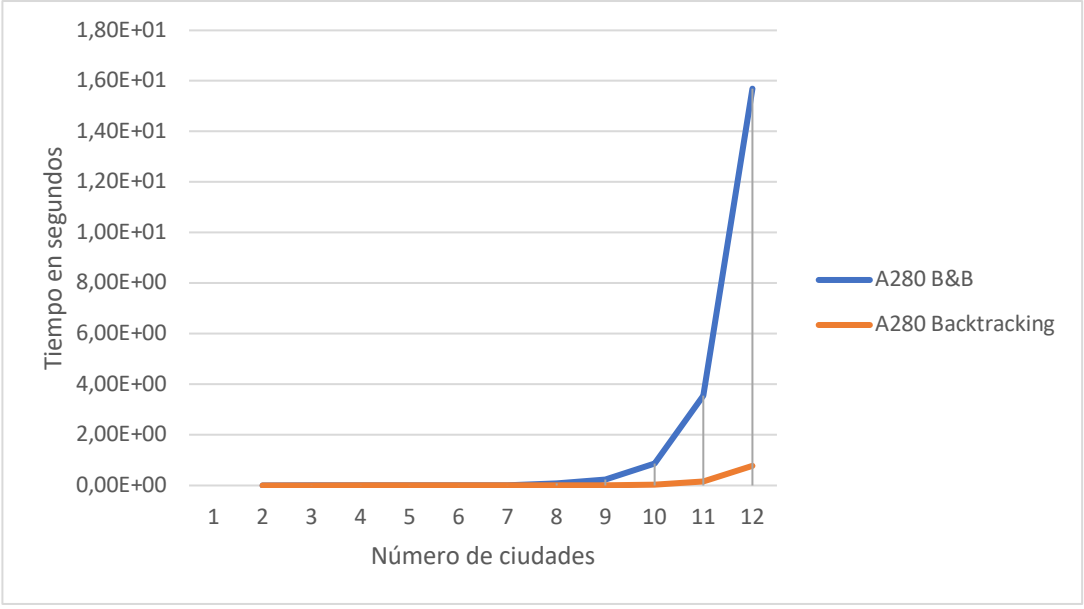
2: Recorrido generado por el algoritmo

DISTANCIA: 204.876
TOUR: 1 5 6 7 8 10 11 12 9 4 3 2 1
NODOS TOTALES: 479001600
NODOS PODADOS: 478101109
NODOS EXPLORADOS: 900491
TIEMPO (seg): 0.777013

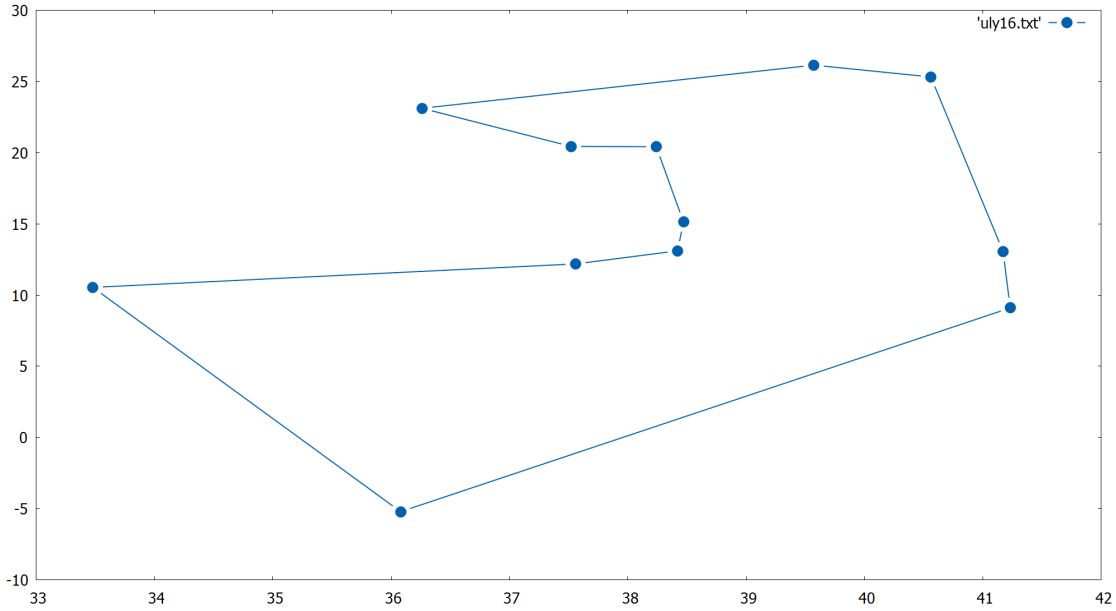
4. Backtracking

DISTANCIA: 204.876
TOUR: 1 5 6 7 8 10 11 12 9 4 3 2 1
NODOS TOTALES: 479001600
NODOS EXPLORADOS: 146053
NODOS EXPANDIDOS: 451766
NODOS PODADOS: 478855547
TIEMPO (seg): 15.6944

5. Branch & Bound



2.4.3. ulysses16



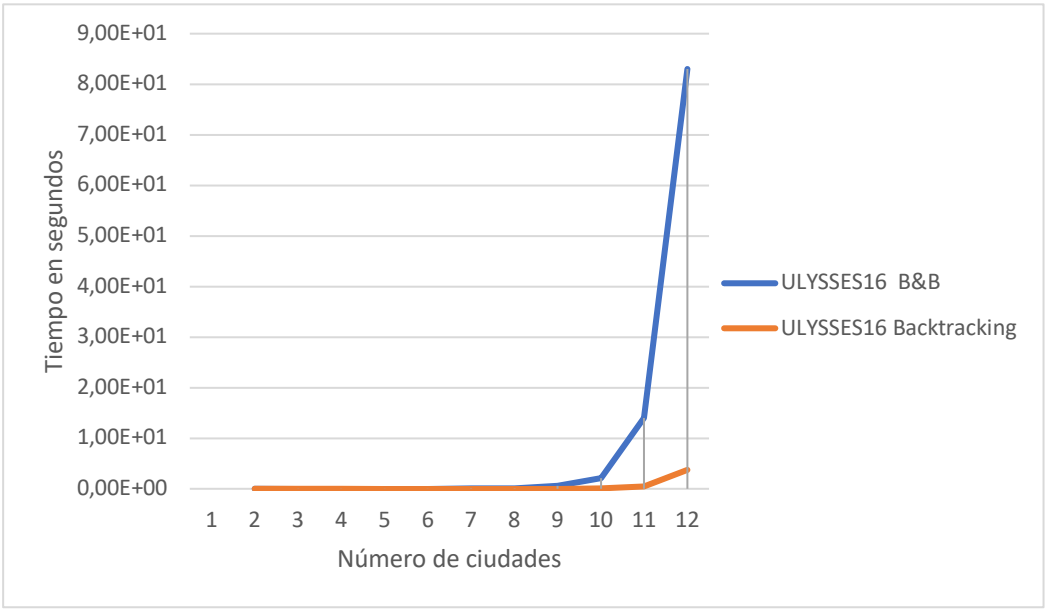
3: Recorrido generado por el algoritmo

DISTANCIA: 69.8443
TOUR: 1 12 7 6 5 11 9 10 3 2 4 8 1
NODOS TOTALES: 479001600
NODOS PODADOS: 474474230
NODOS EXPLORADOS: 4527370
TIEMPO (seg): 3.74082

6. Backtracking

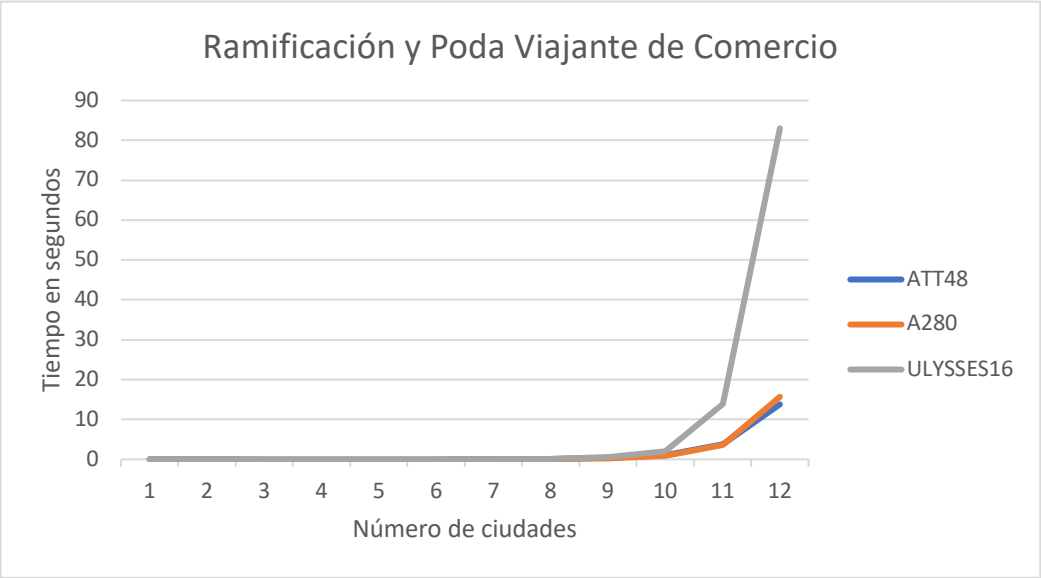
DISTANCIA: 69.8443
TOUR: 1 12 7 6 5 11 9 10 3 2 4 8 1
NODOS TOTALES: 479001600
NODOS EXPLORADOS: 716897
NODOS EXPANDIDOS: 2233532
NODOS PODADOS: 478284703
TIEMPO (seg): 83.0159

7. Branch & Bound

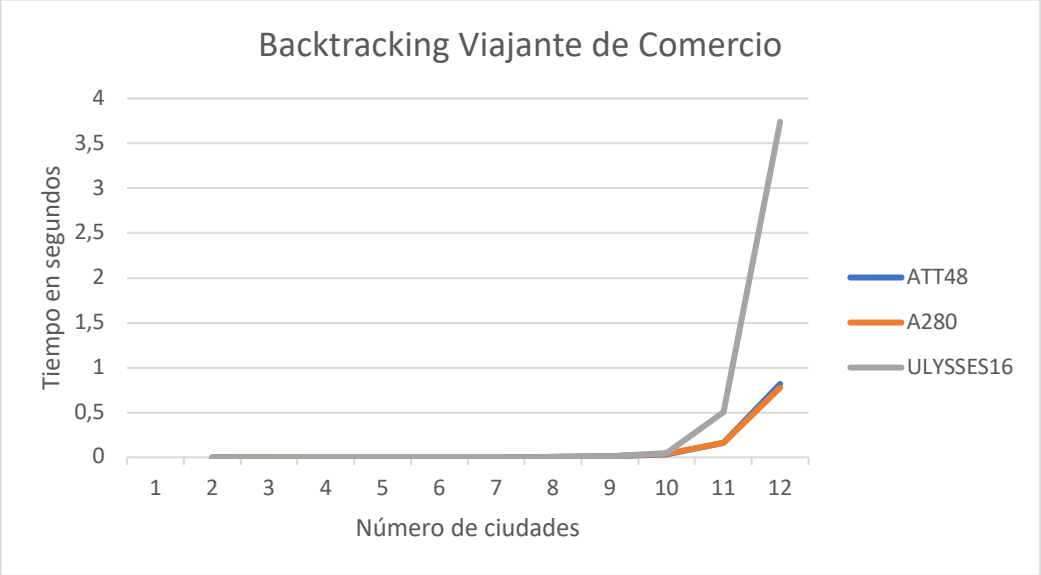


B&B				Backtracking			
ATT48 B&B		A280 B&B		ATT48 Backtracking		A280 Backtracking	
TIEMPO EN SEGUNDOS		TIEMPO EN SEGUNDOS		TIEMPO EN SEGUNDOS		TIEMPO EN SEGUNDOS	
Nº CIUDADES	i7-10510U	Nº CIUDADES	i7-10510U	Nº CIUDADES	i7-10510U	Nº CIUDADES	i7-10510U
1		1		1		1	
2	1,80E-05	2	1,70E-05	2	1,00E-06	2	2,00E-06
3	5,20E-05	3	5,00E-05	3	3,00E-06	3	3,00E-06
4	2,13E-04	4	2,18E-04	4	8,00E-06	4	9,00E-06
5	9,87E-04	5	9,64E-04	5	2,50E-05	5	3,10E-05
6	4,66E-03	6	4,08E-03	6	1,05E-04	6	1,15E-04
7	1,95E-02	7	1,92E-02	7	3,99E-04	7	5,51E-04
8	8,08E-02	8	7,23E-02	8	1,93E-03	8	2,58E-03
9	3,72E-01	9	2,29E-01	9	1,21E-02	9	1,10E-02
10	1,03E+00	10	8,68E-01	10	3,41E-02	10	3,92E-02
11	3,79E+00	11	3,56E+00	11	1,61E-01	11	1,64E-01
12	1,38E+01	12	1,57E+01	12	8,19E-01	12	7,77E-01

ULYSSES16 B&B		ULYSSES16 Backtracking	
TIEMPO EN SEGUNDOS		TIEMPO EN SEGUNDOS	
Nº CIUDADES	i7-10510U	Nº CIUDADES	i7-10510U
1		1	
2	1,70E-05	2	2,00E-06
3	5,20E-05	3	3,00E-06
4	2,21E-04	4	7,00E-06
5	1,05E-03	5	2,30E-05
6	4,52E-03	6	9,90E-05
7	2,21E-02	7	3,40E-04
8	9,99E-02	8	1,74E-03
9	5,27E-01	9	1,12E-02
10	2,06E+00	10	4,85E-02
11	1,40E+01	11	5,05E-01
12	8,30E+01	12	3,74E+00



Vemos cómo el caso de ejecución *Ulysses16* tiene un tiempo de ejecución bastante más alto que el *att48* y el *a280* usando el método Ramificación y Poda.



De igual forma, en el caso de usar el método Backtracking, observamos lo dicho en el párrafo anterior.

2.4.4. Descripción de resultados

A continuación, explicaremos los resultados que hemos obtenido tras la ejecución de estos tres casos.

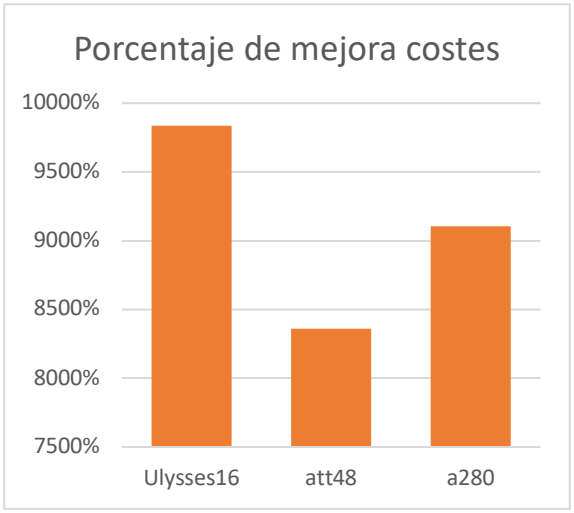
En todos los casos, el uso del algoritmo de backtracking para resolver el problema tiene un tiempo de ejecución mucho menor al algoritmo de Ramificación

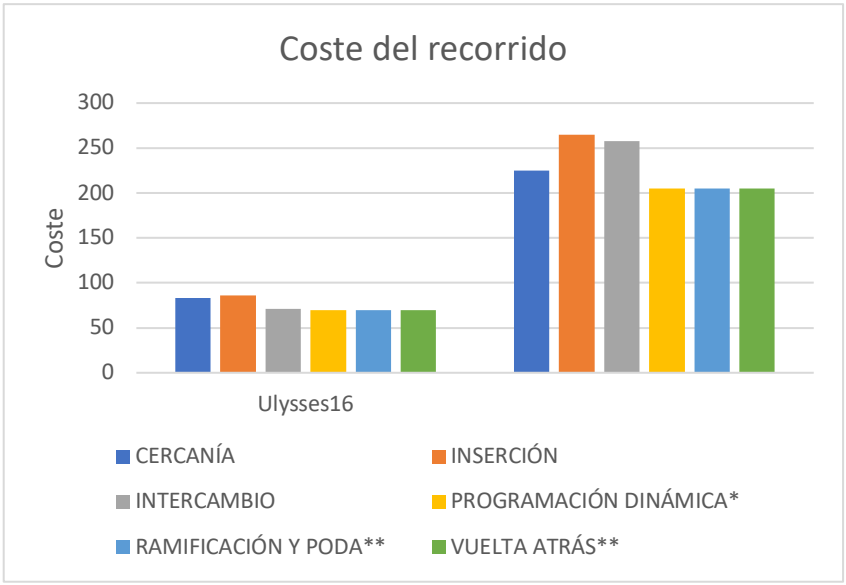
y Poda. Además, el algoritmo backtracking explora muchos nodos más que el de Ramificación y Poda. Queda constancia que el algoritmo backtracking es más eficiente que el de Ramificación y Poda.

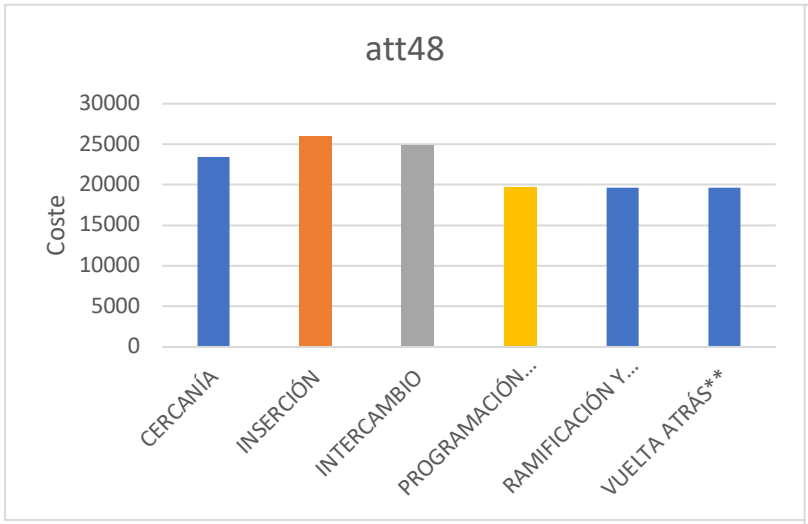
2.5. Comparación con otros métodos

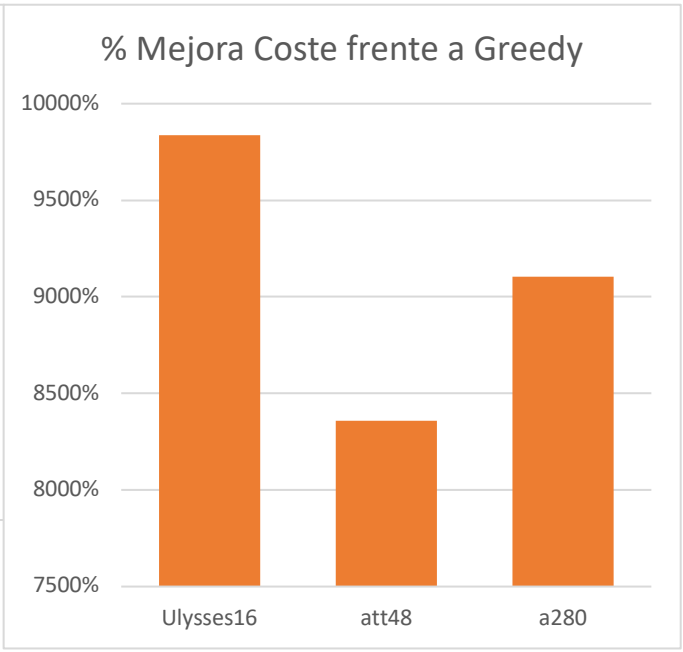
A continuación, expondré una tabla comparativa, tanto en tiempo de ejecución como en coste, con los algoritmos Greedy, programación dinámica y por último Ramificación y Poda y Vuelta Atrás usando los cuatro casos que hemos visto a lo largo de este curso.

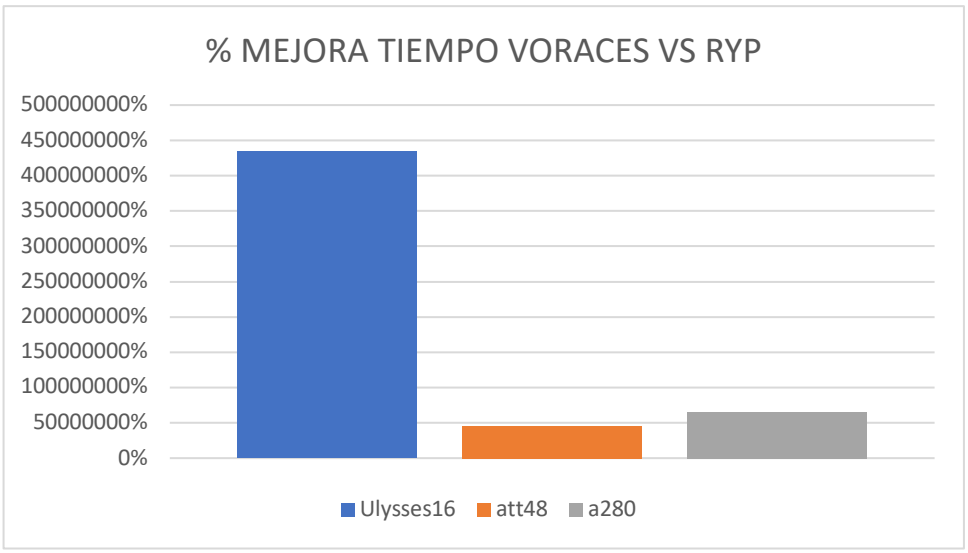
TIEMPO										
	CERCANÍA	INSERCIÓN	INTERCAMBIO	PROGRAMACIÓN DINÁMICA*	RAMIFICACIÓN Y PODA**	VUELTA ATRÁS**	% MEJORA VORACES VS RYP	% MEJORA VORACES VS V.A.	% MEJORA PROG. DINÁMICA VS. RYP	% MEJORA PROG. DINÁMICA VS V.A.
Ulysses16	0,000024	0,000015	0,000081	0,070782	65,2494	0,96732	434996000%	6448800%	92183,6%	1366,6%
att48	0,000024	0,000036	0,000095	0,067714	10,8885	0,213248	45368750,0%	888533,3%	16080,1%	314,9%
a280	0,000032	0,000019	0,000081	0,066633	12,4194	0,199173	65365263,2%	1048278,9%	18638,5%	298,9%
hasta 12 nodos										
COSTE										
	CERCANÍA	INSERCIÓN	INTERCAMBIO	PROGRAMACIÓN DINÁMICA*	RAMIFICACIÓN Y PODA**	VUELTA ATRÁS**	% MEJORA VORACES VS RYP	% MEJORA VORACES VS V.A.	% MEJORA PROG. DINÁMICA VS. RYP	% MEJORA PROG. DINÁMICA VS V.A.
Ulysses16	83	86	71	69,84	69,84	69,84	9837%	9837%	100,00%	100,00%
att48	23466	25977	24860	19614,60	19614,60	19614,60	8358,7%	8358,7%	100,00%	100,00%
a280	225	265	258	205	205	205	9105,6%	9105,6%	100,00%	100,00%
*= hasta 22 nodos										

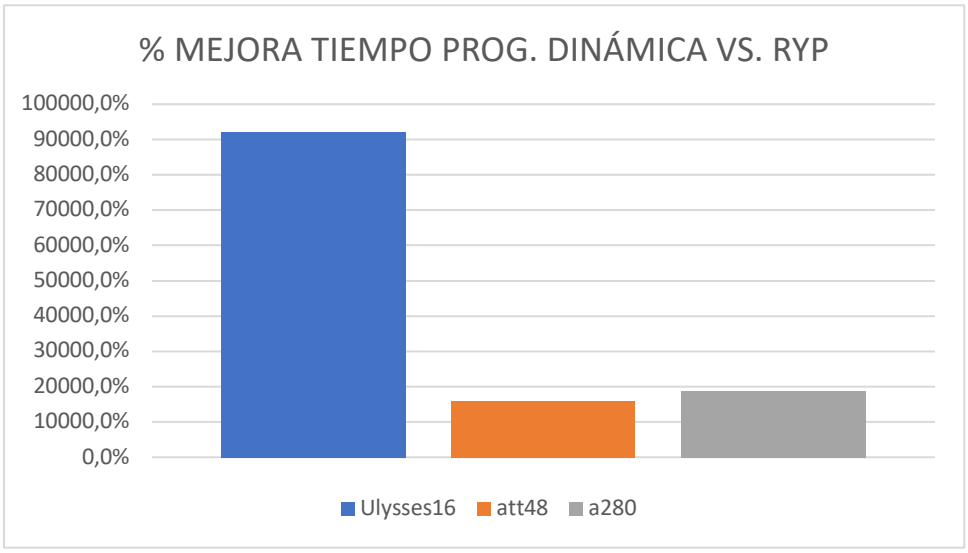





att48
Coste
30000
25000
20000
15000
10000
5000
0
CERCANÍA
INSERCIÓN
INTERCAMBIO
PROGRAMACIÓN...
RAMIFICACIÓN Y...
VUELTA ATRÁS**


% Mejora Coste frente a Greedy
10000%
9500%
9000%
8500%
8000%
7500%
Ulysses16
att48
a280


% MEJORA TIEMPO VORACES VS RYP
500000000%
450000000%
400000000%
350000000%
300000000%
250000000%
200000000%
150000000%
100000000%
50000000%
0%
Ulysses16
att48
a280


% MEJORA TIEMPO PROG. DINÁMICA VS. RYP
100000,0%
90000,0%
80000,0%
70000,0%
60000,0%
50000,0%
40000,0%
30000,0%
20000,0%
10000,0%
0,0%
Ulysses16
att48
a280

Capítulo 3: Conclusión

Para finalizar, tras haber diseñado los tres algoritmos que resuelven tanto el problema de "Cena de Gala", usando el método de vuelta atrás, y el problema del "Viajante de Comercio", usando el método de ramificación y poda, junto con una alternativa usando el método vuelta atrás, haber obtenido los tiempos de ejecución para 3 computadores que poseen especificaciones distintas, y habiendo comparado dichos tiempos, concluimos con haber aprendido cómo el método vuelta atrás es un algoritmo el cual nos asegura que siempre se va a encontrar una solución correcta, siempre que sea un problema el cual tenemos información incompleta, que una opción nos lleva a otro nuevo conjunto de opciones, cuando nos piden el máximo o el mínimo de algo y/o alguna secuencia de decisiones es la solución al problema

Además, hemos visto cómo podemos obtener una solución óptima para el problema del Viajante de Comercio. Abordándolo mediante el método de Ramificación y Poda, hemos visto cómo es una opción más para solucionar este problema, sin embargo, no nos asegura tener un buen comportamiento en cuanto a eficiencia, ya que en el peor caso tiene un tiempo exponencial. Podemos usar distintas técnicas heurísticas para mejorar el tiempo de ejecución. Usando nuestra técnica, LC o "más prometedor", obtenemos un orden de eficiencia mucho mayor comparado con el uso del método Vuelta Atrás y la misma solución.