Universidad de Granada

Algorítmica

Algoritmos Backtracking y Ramificación y Poda, parte 2

Laura Calle Caraballo
Cristina María Garrido López
Germán González Almagro
Javier León Palomares
Antonio Manuel Milán Jiménez

7 de junio de 2016

Índice

1. Introducción.

El objetivo de esta práctica es el estudio de las técnicas de tipo *Ramificación y Poda* y *Backtracking*, aplicadas particularmente al problema del viajante de comercio. Para ello, hemos implementado ambos algoritmos y, adicionalmente, hemos probado algunas modificaciones sobre el primero de ellos.

2. Descripción del problema.

El problema del viajante de comercio, también llamado *TSP* por sus siglas en inglés, es conocido desde el siglo XIX y fue planteado en su forma general en la década de 1930; desde entonces, es uno de los problemas más estudiados en optimización. Su complejidad hace computacionalmente muy costosa una solución mediante fuerza bruta. Por ello, se han ido creando una serie de métodos que obtienen resultados válidos según la situación: soluciones exactas para dimensiones pequeñas, aproximadas para dimensiones más grandes o particularizaciones del problema donde se pueda emplear una aproximación mejor.

La pregunta que busca responder este problema es la siguiente: *"Considerando un conjunto de ciudades y las distancias entre ellas dos a dos, ¿cuál es el camino más corto que pasa por todas ellas una única vez y retorna al origen?"*

En términos de grafos esto significa que, a partir de un grafo ponderado, conexo y no dirigido, debemos encontrar el circuito hamiltoniano de menor peso.

3. Resolución.

Para afrontar la resolución de este problema de forma exacta, hemos implementado un algoritmo de *Ramificación y Poda*, también denominado *Branch and Bound*. Las características de esta técnica particularizadas para el problema del viajante de comercio son las siguientes:

- Solución: vector que contiene todas las ciudades de la instancia a evaluar ordenadas según se recorren para formar un circuito hamiltoniano.
- Función de poda: si una solución parcial tiene una cota inferior mayor que el coste de la mejor solución encontrada hasta el momento (cota superior), se realiza una poda.
- Restricciones explícitas: las ciudades almacenadas en la solución pertenecen a la instancia que se está evaluando.
- Restricciones implícitas: una ciudad no puede aparecer más de una vez en la solución (excepto si representamos el cierre del circuito añadiendo de nuevo la ciudad de partida).
- Espacio de soluciones: las permutaciones de tamaño n, siendo n la dimensión del problema.

Según lo anterior, es necesario el uso de dos tipos de cotas: una superior y una inferior. La cota superior representa el coste de la mejor solución conocida por el algoritmo, y será global. La cota inferior se define para cada solución parcial, y representa el coste desde el inicio hasta la última ciudad insertada sumado a una estimación optimista del costo restante. Usando estos valores podemos podar cuando la cota inferior de una solución parcial sea mayor que la cota superior, ya que significa que dicha solución parcial nunca podrá ser mejor que lo que ya tenemos.

Adicionalmente, se ha implementado un algoritmo *Backtracking* que también consta de los elementos antes listados.

3.1. Algoritmo de *Ramificación y Poda*.

3.1.1. Pseudocódigo.

A continuación, explicamos el funcionamiento de esta técnica mediante pseudocódigo:

```
function ViajanteComercio(conjuntoCiudades,ciudadInicial);
mejorCoste ⟵ cotaSuperior[1];
caminoInicial ⟵ ciudadInicial;
abiertos.push(caminoInicial);
begin
    while abiertos not empty do
        caminoActual ⟵ abiertos.top();
        abiertos.pop();
        if EsSolucion(caminoActual) then
            if Coste(caminoActual)[2] < mejorCoste then
                mejorCamino ⟵ caminoActual;
                mejorCoste ⟵ Coste(caminoActual);
            end
        else if Valoracion(caminoActual) < mejorCoste then
            while quedan hijos sin generar do
                hijo ⟵ GeneraHijo(caminoActual);
                if Valoracion(hijo) < mejorCoste then
                    abiertos.push(hijo);
                end
            end
        end
    end
    return mejorCamino, mejorCoste;
end
```

[1] La cota superior inicial puede tomar el valor ∞ o bien el resultado de un algoritmo *Greedy*.

[2] Nótese la diferencia entre las funciones *Coste(camino)* y *Valoracion(camino)*. La primera es usada para caminos completos y no contiene elementos heurísticos; en cambio, la segunda se utiliza para caminos parciales y siempre tiene una componente de estimación de coste. Esto es aplicable también a la siguiente sección.

3.2. Algoritmo *Backtracking*.

A diferencia de la técnica de *Ramificación y Poda*, *Backtracking* no prioriza su exploración por criterios de coste o heurísticos, sino que explora cada rama en profundidad. Sin embargo, podemos reducir la búsqueda introduciendo la función de poda que hemos usado anteriormente.

3.2.1. Pseudocódigo.

A continuación se muestra el pseudocódigo del algoritmo incluyendo la función de poda:

```
ciudadActual ⟵ ciudadInicial;
caminoActual ⟵ caminoActual ∪ ciudadActual;
mejorCoste ⟵ cotaSuperior;
function ViajanteComercio(conjuntoCiudades, caminoActual);
begin
    if EsSolucion(caminoActual) then
        if Coste(caminoActual) < mejorCoste then
            mejorCamino ⟵ caminoActual;
            mejorCoste ⟵ Coste(caminoActual);
        end
    else if Valoracion(caminoActual) < mejorCoste then
        while quedan hijos sin generar do
            hijo ⟵ GeneraHijo(caminoActual);
            caminoActual ⟵ caminoActual ∪ hijo;
            ViajanteComercio(conjuntoCiudades,caminoActual);
        end
    end
    caminoActual ⟵ caminoActual − caminoActual.back();
end
```

4. Mejoras sobre el algoritmo de *Ramificación y Poda*.

En esta sección analizaremos cómo el nivel de optimización al compilar y ciertos cambios en las cotas superiores e inferiores afectan al rendimiento de esta técnica.

4.1. Optimización en tiempo de compilación.

Todas las mediciones incluidas en esta memoria han sido realizadas con el nivel de compilación 2 del compilador *g++*. Una muestra de la diferencia con una compilación normal se puede ver en la siguiente tabla:

Tamaño	**Nivel 0**	**Nivel 2**
9	0.066694	0.005936
11	1.0335	0.097976
13	25.5159	2.39737
15	1192.46	113.248

Figura 1: Tiempos de ejecución de *Ramificación y Poda* con optimizaciones 0 y 2.

4.2. Cota inferior (heurística).

Hemos probado dos heurísticas diferentes, una menos costosa pero menos precisa (heurística 1) y otra más costosa pero más precisa (heurística 2):

- Heurística 1: la valoración de un nodo consiste en el coste real desde el inicio hasta la última ciudad añadida más la suma de la distancia más corta asociada a cada ciudad no contenida en el circuito.
- Heurística 2: similar a la anterior, pero sustituyendo cada distancia más corta asociada a cada ciudad no incluida por la media de las dos distancias más cortas.

Aquí tenemos una tabla comparando los tiempos de ejecución obtenidos:

Tamaño	**Heurística 1**	**Heurística 2**
9	0.005936	0.010186
10	0.025171	0.021891
11	0.097976	0.150797
12	0.40271	0.327216
13	2.39737	1.88105
14	15.5334	13.411
15	113.248	76.734
16	507.988	475.295
17	1403.64	1342.11

Figura 2: Tiempos de ejecución del algoritmo de *Ramificación y Poda* con las dos heurísticas sobre subconjuntos de la instancia *ulysses22*.

La gráfica de tiempos de ejecución es la siguiente:

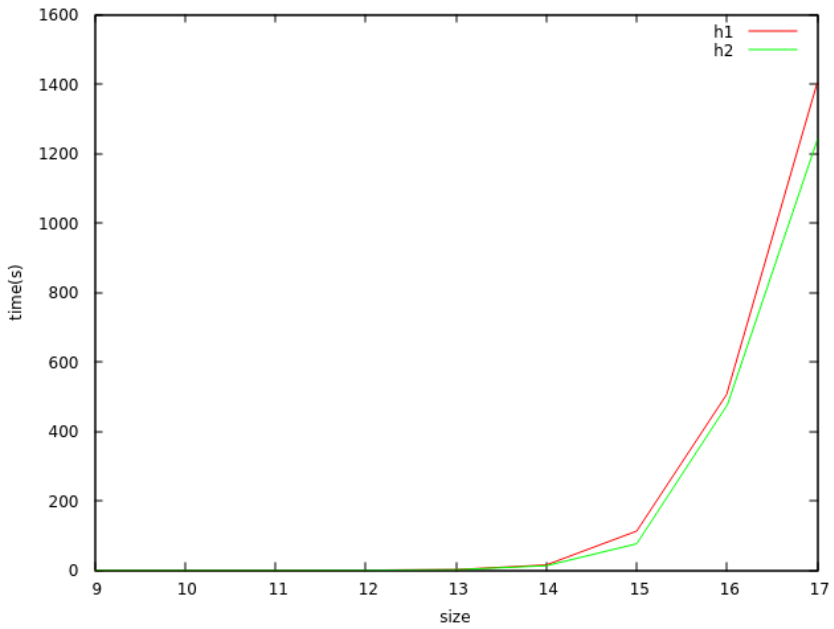

Figura 3: Gráfica comparativa de tiempos de ejecución. Intel® Core^{TM} i5-5200U CPU @ 2.20GHz.

Además, podemos observar diferencias en el número de nodos expandidos, el número de podas realizadas y el tamaño máximo de la lista de nodos abiertos:

Tamaño	Nodos explorados		Podas realizadas		Tamaño máximo de abiertos	
	Heurística 1	**Heurística 2**	**Heurística 1**	**Heurística 2**	**Heurística 1**	**Heurística 2**
9	11835	9318	13931	11037	20	20
10	46104	35096	68824	50800	25	25
11	169904	123479	315454	216650	28	42
12	658812	501201	1411538	996808	32	49
13	3700596	2664686	8897596	5897746	40	98
14	22759953	17305665	59114136	40576941	46	281
15	159973208	98810551	433414136	246071382	52	525
16	747710543	577997680	2075155499	1563321736	62	579
17	2362464162	1744774322	6545043838	5062870607	62	709

Figura 4: Otros datos relevantes de las dos heurísticas.

Como apunte, la cota superior inicial utilizada corresponde a una ejecución *Greedy*.

Como muestra de las diferencias, a continuación mostramos una gráfica con la cantidad de nodos explorados y otra con la cantidad de podas realizadas usando cada heurística:

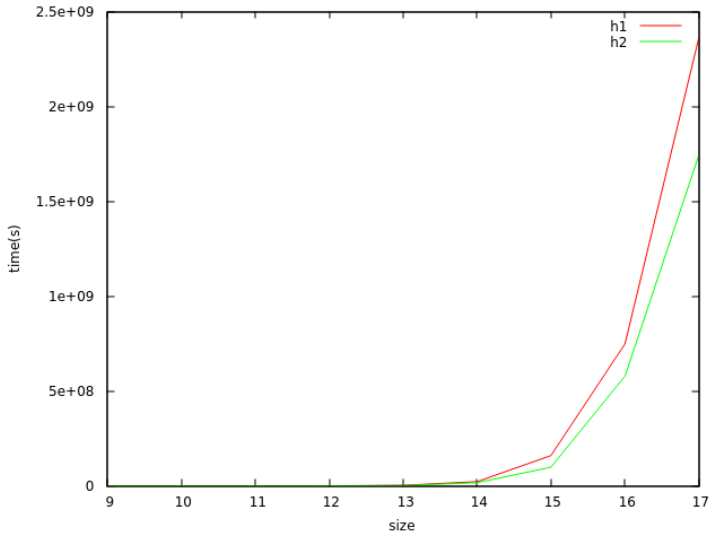

Figura 5: Gráfica comparativa de nodos explorados.

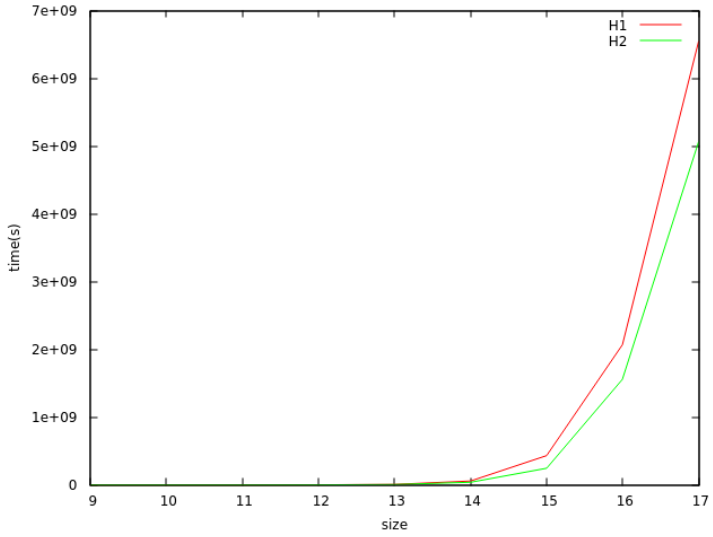

Figura 6: Gráfica comparativa de nodos podados.

Podemos intuir que, gracias a la mayor precisión de la heurística 2, el algoritmo es capaz de realizar más podas en las fases iniciales, disminuyendo tanto el número total de nodos explorados como el de podas, ya que el factor de ramificación se reduce ligeramente.

4.3. Cota superior inicial.

Con el objetivo de reducir el espacio de soluciones desde el inicio, podemos acotar el coste máximo que aceptaremos para que sea estrictamente menor que el resultado de un algoritmo *Greedy* frente a una cota inicial con valor infinito. De esta forma, es posible evitar la comprobación de más permutaciones potencialmente inútiles.

Veamos primero los tiempos de ejecución:

Tamaño	**Infinito**	***Greedy***
9	0.006572	0.005936
10	0.030662	0.025171
11	0.11359	0.097976
12	0.524846	0.40271
13	3.01591	2.39737
14	20.2228	15.5334
15	142.716	113.248
16	596.025	507.988
17	1510.34	1403.64

Figura 7: Tiempos de ejecución de los dos algoritmos sobre subconjuntos de la instancia *ulysses22*.

La gráfica asociada a la tabla anterior es:

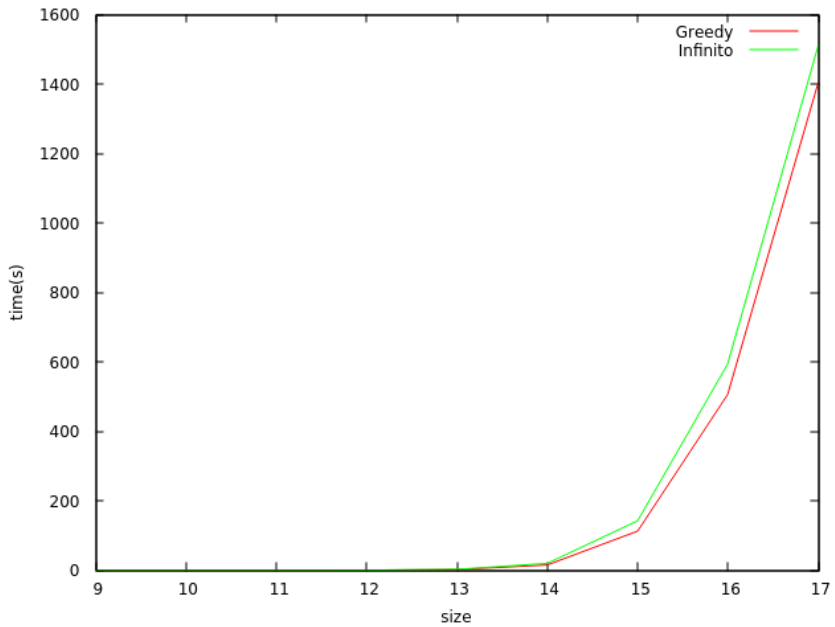

Figura 8: Gráfica comparativa de tiempos de ejecución. Intel® Core™ i5-5200U CPU @ 2.20GHz.

Asimismo, los efectos de esta modificación en otros aspectos importantes se pueden observar en la siguiente tabla:

Tamaño	Nodos explorados		Podas realizadas		Tamaño máximo de abiertos	
	Infinito	***Greedy***	**Infinito**	***Greedy***	**Infinito**	***Greedy***
9	14620	11835	16127	13931	29	20
10	55346	46104	77266	68824	37	25
11	211088	169904	369805	315454	46	28
12	917049	658812	1859356	1411538	56	32
13	4898698	3700596	11313246	8897596	68	40
14	30180661	22759953	75441404	59114136	81	46
15	305703379	159973208	547508027	433414136	94	52
16	983795806	747710543	2478489330	2075155499	109	62
17	3396325251	2362464162	6488569310	3545043838	123	62

Figura 9: Otros datos relevantes de los dos valores iniciales de cota superior.

Asimismo, mostramos las gráficas correspondientes a los nodos explorados y podados según las dos cotas superiores iniciales:

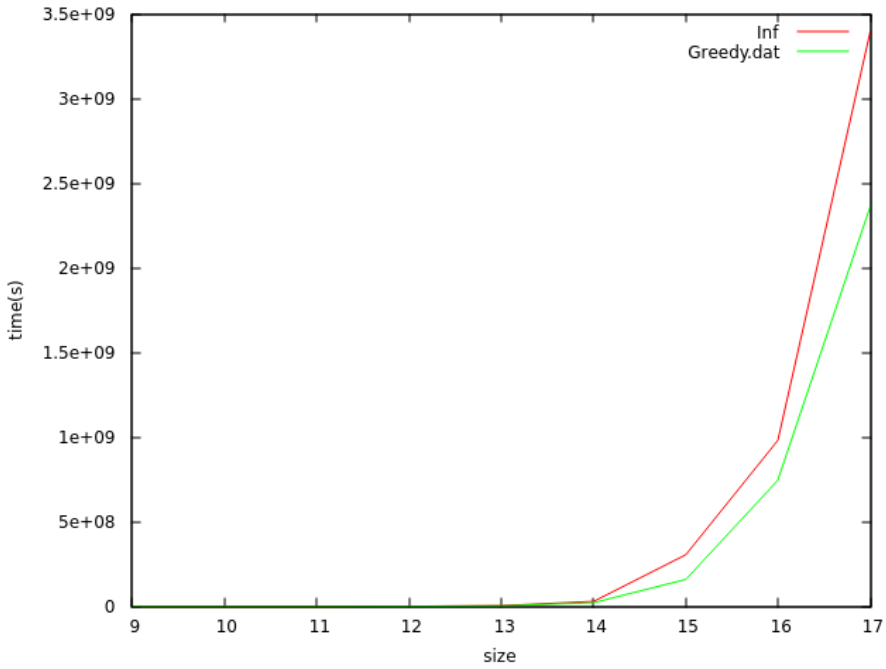

Figura 10: Gráfica comparativa de nodos explorados.

En la gráfica anterior podemos observar cómo el disponer de una cota superior inicial más realista permite descartar muchas posibilidades desde el principio.

Como consecuencia, el factor de ramificación se reduce y, con él, el número de nodos que es necesario podar en fases posteriores de la ejecución. Esto se comprueba aquí:

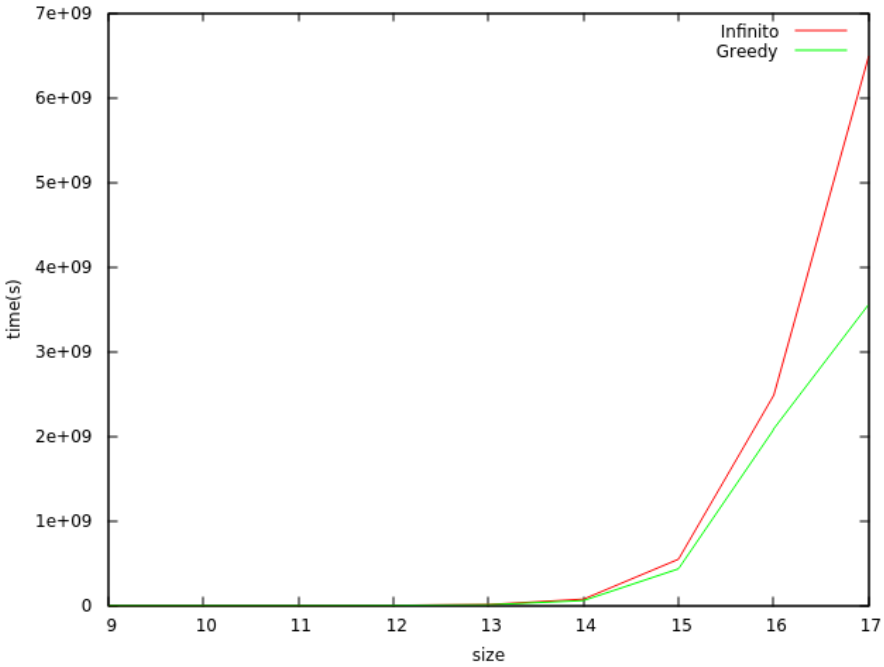

Figura 11: Gráfica comparativa de nodos podados.

5. Comparativa entre *Ramificación y Poda* y *Backtracking*.

En este apartado vamos a comparar ambos algoritmos bajo las mismas condiciones según el tiempo consumido y el esfuerzo de exploración del espacio de búsqueda realizado. Para ello, se ha utilizado como criterio de poda la heurística 1 y como cota superior inicial una solución *Greedy*; para más detalles sobre estos componentes, ver la **sección anterior**.

En primer lugar, presentamos los tiempos de ejecución de ambos algoritmos mediante una tabla y su gráfica asociada:

Tamaño	Ramificación y Poda	Backtracking
9	0.005936	0.004954
10	0.025171	0.019709
11	0.097976	0.079026
12	0.40271	0.409318
13	2.39737	2.62488
14	15.5334	15.9302
15	113.248	142.929
16	507.988	553.935
17	1403.64	1732.71

Figura 12: Tiempos de ejecución de los dos algoritmos sobre subconjuntos de la instancia *ulysses22*.

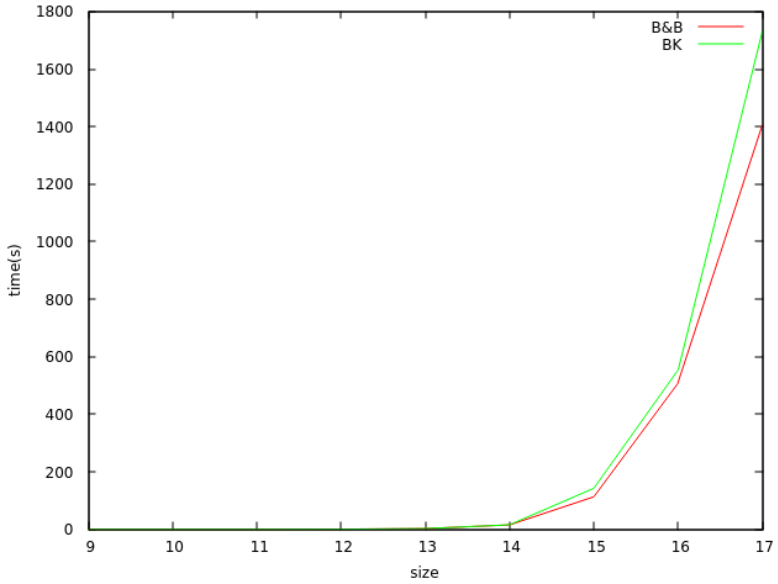

Figura 13: Gráfica comparativa de tiempos de ejecución. Intel® Core™ i5-5200U CPU @ 2.20GHz.

Sin embargo, el tiempo de ejecución no es el único dato relevante a la hora de comparar estas técnicas. Para ilustrar mejor las diferencias, podemos compararlas también según el número total de nodos explorados y el número de podas realizadas:

Tamaño	Nodos explorados		Podas realizadas	
	Ramificación y Poda	**Backtracking**	**Ramificación y Poda**	**Backtracking**
9	11835	24879	13931	11433
10	46104	100544	68824	55162
11	169904	386177	315454	239130
12	658812	1883271	1411538	1230383
13	3700596	11417664	8897596	7116013
14	22759953	66055391	59114136	46618259
15	159973208	555393772	433414136	396271023
16	747710543	2551655421	2075155499	1541858327
17	2362464162	3612963039	6545043838	3079201741

Figura 14: Otros datos relevantes de los dos algoritmos.

La diferencia en número de nodos explorados y podados se ve en las siguientes dos gráficas:

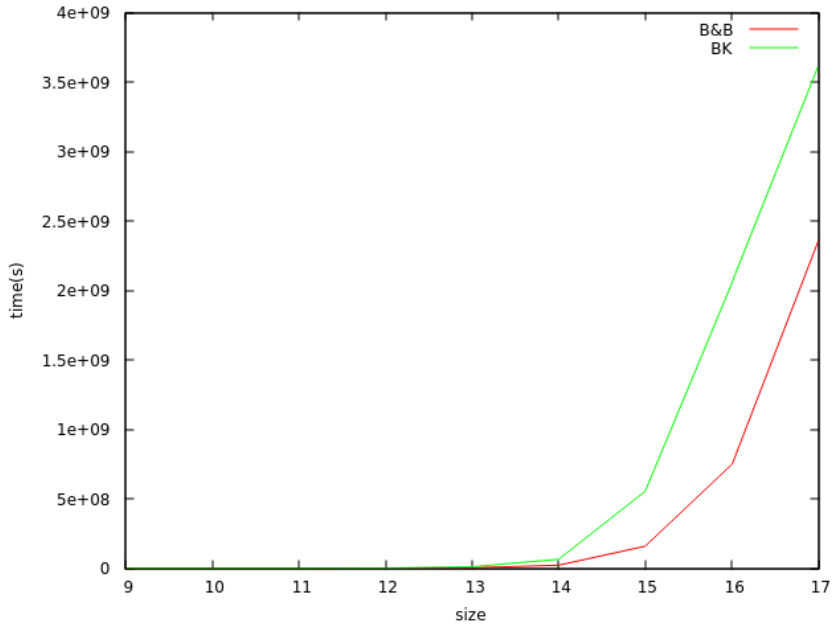

Figura 15: Gráfica comparativa de nodos explorados.

Como podemos observar, *Ramificación y Poda* explora muchos menos nodos que *Backtracking*.

Y, sin embargo, también es capaz de podar una mayor cantidad de nodos.

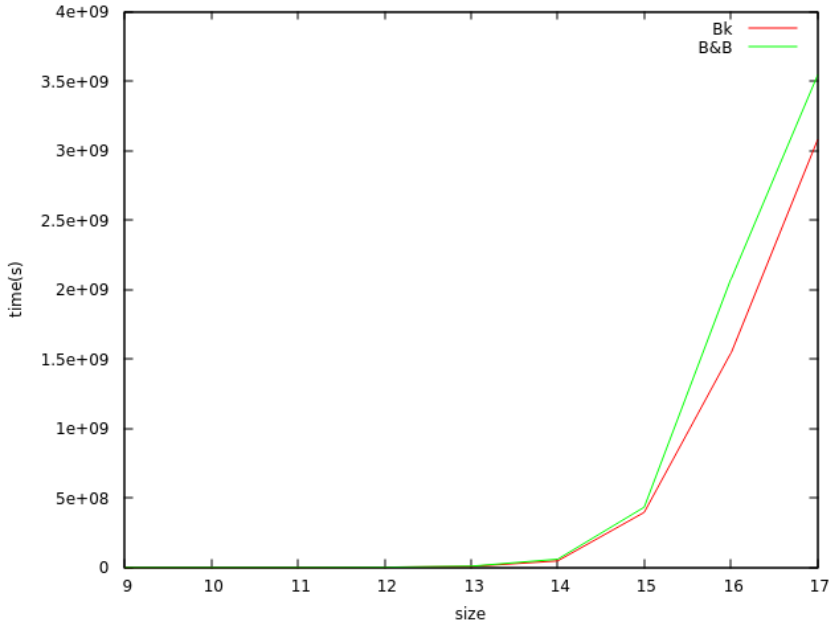

Figura 16: Gráfica comparativa de nodos podados.

6. Conclusión.

A la vista de los resultados de la práctica, podemos decir que un técnica de *Ramificación y Poda* es mejor que una técnica *Backtracking* para resolver de forma óptima el problema del viajante de comercio, excepto en aquellos casos en los que la carga que supone manejar las estructuras de datos necesarias hace que *Ramificación y Poda* emplee más tiempo en resolver el problema (esto sucede para volúmenes de datos pequeños).

Es destacable también la elección de una buena heurística y la forma de determinar la cota superior inicial, ya que influyen de manera notable en la eficiencia del algoritmo. Cuanto más se ajusten ambas a la realidad, más eficiente será nuestro algoritmo.

Otra forma de disminuir los tiempos de ejecución es la optimización del compilador. Esto nos ha permitido realizar pruebas con dimensiones un poco más grandes, por lo cual concluimos que es una opción a tener en cuenta.

En cuanto a cuestiones de implementación, debemos tener cuidado a la hora de elegir la representación interna de los datos, puesto que, por ejemplo, un dato de tipo entero no es capaz de almacenar la suma de los nodos explorados o expandidos para volúmenes de datos considerables; en ese caso, sería necesario un tipo de dato con mayor capacidad. Asimismo, no hay que despreciar el orden de eficiencia del cálculo de la valoración optimista de cada nodo, puesto que es un cálculo que puede llegar a realizarse millones de veces y, por tanto, influye notablemente en el tiempo de ejecución del algoritmo.