UNIVERSIDAD DE GRANADA

Práctica 5: Bactracking Y B&B

Análisis de algoritmos Bactracking y Branch & Bound

José María Gómez García
Fernando Lojano Mayaguari
Valentino Lugli
Carlos Mulero Haro

Universidad de Granada
Granada
Junio 2020

Índice

1 Introducción **2**

2 Problema: ITV **3**
2.1 Descripción de la solución . . . 3
2.2 Pseudocódigo . . . 3
2.3 Análisis empírico . . . 4
2.3.1 Para m=2 . . . 4
2.3.2 Para m=3 . . . 5
2.4 Escenarios de ejecución . . . 7

3 Problema: TSP **9**
3.1 Descripción de la solución . . . 9
3.2 Pseudocódigo . . . 9
3.3 Escenarios de ejecución . . . 11
3.4 Comparación entre B&B y Backtracking . . . 13

4 Conclusión **15**

1 Introducción

En esta quinta y última práctica vamos a analizar algoritmos con enfoque Bactracking y Branch & Bound. Para empezar, llevaremos a cabo una pequeña explicación acerca del funcionamiento del enfoque de Backtracking y posteriormente se explicará el enfoque Branch & Bound. Se explicarán ambos algoritmos en este orden debido a que el enfoque Branch & Bound es una variante avanzada del Backtracking.Dicho esto, comenzaremos con Backtracking:

La técnica Backtracking o también conocida como la técnica de vuelta atrás es un método basado en la ramificación que consiste en buscar un conjunto solución analizando todas las ramas de un grafo, expandiendo todos los nodos del mismo, obteniendo como resultado una solución óptima. Podemos llegar a la conclusión por tanto que Backtracking no es eficiente en tiempo de ejecución. Esto se debe a que, a diferencia de los algoritmos voraces (por ejemplo) que no eran capaces de deshacer una decisión una vez que haya sido tomada en la solución parcial de un problema, Backtracking analiza la solución parcial junto con sus sucesivas decisiones para el resto de soluciones parciales y, después vuelve a tener en cuenta el mismo subproblema escogiendo una solución diferente, analizando así todas las combinaciones existentes en ese subproblema.

Si bien ya sabemos que este enfoque no es eficiente en términos de tiempo, al aplicar alguna técnica de ramificación y poda, el tiempo conseguido para obtener una solución óptima es mucho mejor que el únicamente usando el enfoque de Backtracking. Esta técnica es la segunda que vamos a analizar en esta práctica y que denominamos como Branch & Bound.

La diferencia principal entre la primera técnica y la segunda es que, si imaginamos un grafo con todos los nodos, Backtracking realiza un recorrido primero en profundidad expandiendo los hijos del primer nodo(todos) y, después de haberlos analizado, escogemos los hermanos del primer nodo expandiendo también sus respectivos hijos. Esto no tiene por que ser así usando la técnica de ramificación y poda, ya que dependiendo de la forma en la que calculemos el beneficio estimado, escogeremos un nodo diferente y no el primero que nos encontremos. Además, el tiempo también se ve mejorado debido a que usamos cotas para realizar una poda y evitar el análisis de conjuntos de soluciones cuyo resultado no es óptimo. Dependiendo de si se trata de un problema de maximización o minimización usaremos una cota superior o inferior.

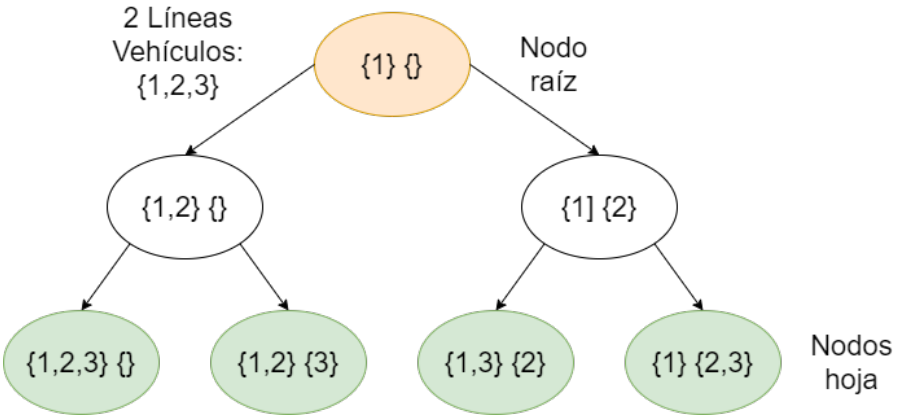
Una vez explicados ambos enfoques de una forma básica, pasaremos a analizar el funcionamiento de estas técnicas en dos problemas, uno de ellos es el problema asignado llamado ITV y el segundo es el ya conocido problema del viajante del comercio o TSP.

2 Problema: ITV

2.1 Descripción de la solución

En el problema ITV queremos representar una estación de ITV que conste de varias líneas en las que se inspeccionan vehículos. Cada coche tendrá distintas características por lo que el tiempo de inspección varía según el vehículo. El objetivo del problema es atender a todos los vehículos en el menor tiempo posible.

Para dar con una solución usando un algoritmo de vuelta atrás empezamos añadiendo un vehículo cualquiera a la primera línea (este será nuestro nodo raíz); a partir de este expandimos el árbol añadiendo el siguiente vehículo en cola a cada línea (cada vez que añadimos el siguiente vehículo en cola a una línea expandimos en un nodo); cuando no queden coches en cola quiere decir que hemos dado con una solución, para quedarnos con la solución que acabe en el menor tiempo posible seleccionamos aquella cuya línea con mayor tiempo de espera sea la menor de entre todas las líneas con mayor tiempo de espera de las soluciones.



Ejemplo de la estructura de árbol creada durante una ejecución

2.2 Pseudocódigo

```
// coches_encola contiene los coches que todavia no estan en ninguna linea
// coches_listos contiene los coches en cada una de las lineas, por ejemplo,
   coches_listos[l][c] estaria accediendo al coche c de la linea l. Podemos
   empezar a resolver con un coche ya en la primera linea

funcion resolver_itv(choches_encola, coches_listos)

    if (!coches_encola.vacio)
        for i=0 while i < lineas
            coches_encola.push_back(coches_listos[0])
            coches_listos.erase(0)
            resolver(coches_encola, coches_listos)
        fin
    fi
    // No quedan coches en cola, todos estan listos, es un nodo hoja
    else
        tiempo_linea_max := int
        // Calculamos el tiempo en cada cola y nos quedamos con el mayor tiempo
            de espera
        for i=0 while i < lineas
            tiempo_linea[i] := tiempo(coches_listos[i])
            if tiempo_linea_max es menor que tiempo_linea[i] entonces
                tiempo_linea_max := tiempo_linea[i]
            fi
```

```
        fin
        // Si el mayor tiempo de espera es menor que la solucion que teniamos
           anteriormente este pasa a ser la solucion
        if tiempo_linea_max es menor que el tiempo_solucion entonces
            tiempo_solucion := tiempo_linea_max
            solucion := coches_listos
        fi
    esle

function
```

2.3 Análisis empírico

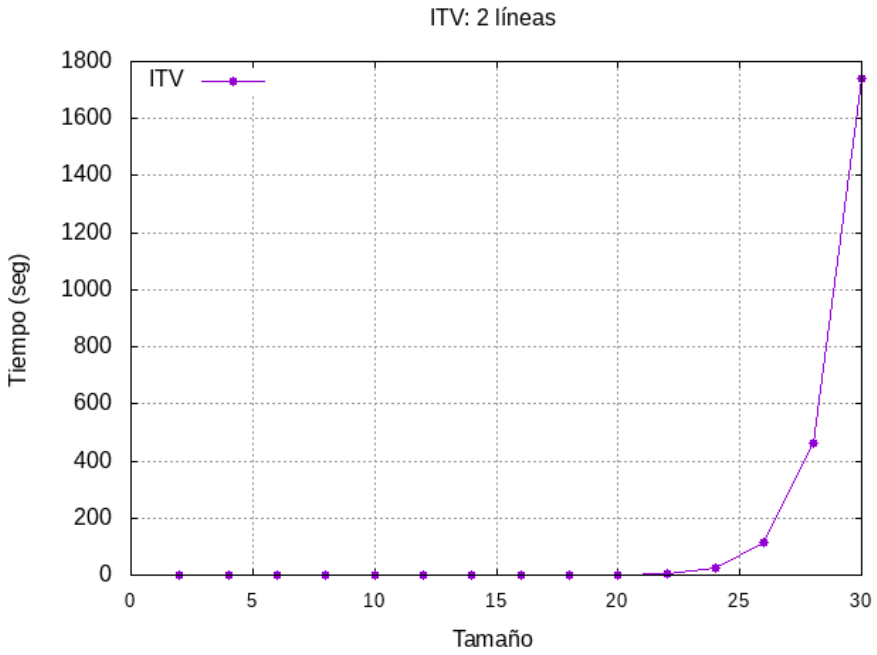
Para realizar el análisis empírico del problema, vamos a comentar dos gráficas obtenidas de poner un valor de m (número de lineas) de inspección fijo. En nuestro caso hemos hecho una gráfica para m=2 y para m=3.

2.3.1 Para m=2

Para el algoritmo de la ITV con dos lineas hemos realizado un script que introduce los datos en el rango de 2 a 30 coches con un incremento de 2 en cada iteración. Una vez sacados los datos, obtenemos esta tabla:

Número de coches	Tiempo (seg)
2	1.9e-05
4	7.2e-05
6	0.000275
8	0.001118
10	0.004405
12	0.01759
14	0.034859
16	0.105486
18	0.401111
20	1.59577
22	6.68534
24	26.0182
26	113.879
28	462.951
30	1739.29

Con los datos de la tabla obtenemos la siguiente gráfica:



Gráfica de tiempos de ejecución para m = 2

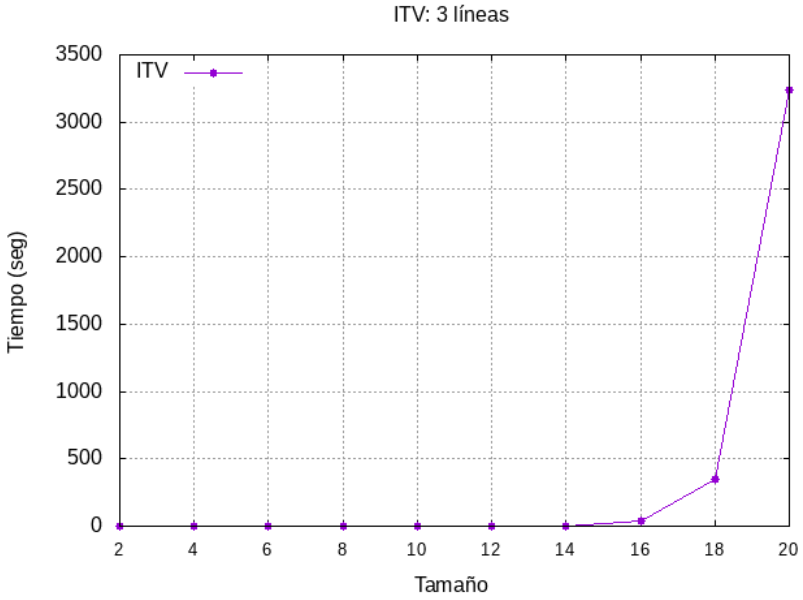
Cabe destacar que en las primeras ejecuciones el incremento no es muy apreciable. Por ello en este caso hemos tomado valores hasta 30, ya que así podemos ver fácilmente el incremento de tiempo.

2.3.2 Para m=3

Para el algoritmo de la ITV con dos lineas hemos realizado un script que introduce los datos en el rango de 2 a 20 coches con un incremento de 2 en cada iteración. Una vez sacados los datos obtenemos esta tabla:

Número de coches	Tiempo (seg)
2	9e-06
4	7.3e-05
6	0.000631
8	0.005696
10	0.051884
12	0.469677
14	4.27036
16	38.7651
18	351.399
20	3240.74

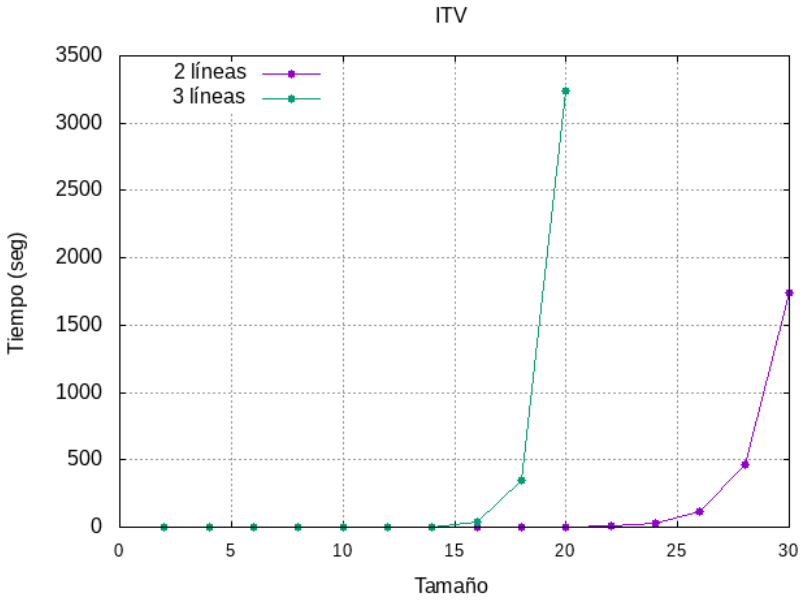
Con los datos de la tabla obtenemos la siguiente gráfica:



Gráfica de tiempos de ejecución para m = 3

En este caso no tomamos tantos valores ya que para 3 lineas apreciamos antes el crecimiento de la gráfica.

Gráfica de tiempos de ejecución conjunta



Al observar las dos gráficas conjuntamente, vemos que ambas a partir de cierto valor (diferente en cada curva) comienza a crecer de forma más brusca. Esto nos hace pensar que el comportamiento del algoritmo de la ITV presenta una eficiencia de orden exponencial.

2.4 Escenarios de ejecución

Para realizar las pruebas de este programa se ha utilizado un banco de datos con varios automóviles y sus respectivos tiempos, para cada una de las pruebas únicamente se ha ido aumentando el rango de datos escogidos, es decir, si probamos con 2 automóviles, tomamos solo los 2 primeros automóviles, si escogemos 8 automóviles, escogemos los 8 primeros automóviles. Se repite el proceso sucesivamente hasta llegar, en el caso de dos líneas a 30 automóviles y en el caso de tres líneas a 20.

Automóvil	Tiempo	Automóvil	Tiempo (seg)
1	4	16	7
2	7	17	1
3	18	18	7
4	16	19	13
5	14	20	17
6	16	21	12
7	7	22	9
8	13	23	8
9	10	24	10
10	2	25	3
11	3	26	11
12	8	27	3
13	11	28	4
14	20	29	8
15	4	30	16

El tiempo total acumulado de todos estos automóviles es 282. Otro aspecto a destacar es que antes de comenzar a ejecutar la función se introduce el último automóvil del vector de automóviles en la última linea de las que disponemos. Esto significa que, en el caso de tener dos líneas, se introduce siempre antes de comenzar la función de resolución el último automóvil disponible en la segunda línea y en el caso de tres líneas, el último automóvil se introduce en la tercera línea. A continuación se van a mostrar los resultados generados para cada uno de los casos, comenzando por el caso en el que disponemos de dos líneas:

Tamaño	**T L1**	**T L2**	**T Sol**	**Línea 1**	**Línea 2**
2	4	7	7	1	2
4	22	23	23	3-1	4-2
6	37	38	38	5-4-2	6-3-1
8	48	47	48	7-6-5-2-1	8-4-3–
10	53	54	54	9-8-7-6-2	10-5-4-3-1
12	59	59	59	11-10-9-7-6-5-2	12-8-4-3-1
14	74	75	75	13-12-11-10-9-8-7-6-1	14-5-4-3-2
16	80	80	80	15-14-13-12-11-10-9-3-1	16-8-7-6-5-4-2
18	84	84	84	16-15-14-13-12-11-10-9-7-2-1	18-8-6-5-4-3
20	99	99	99	19-18-17-16-15-14-13-11-11-10-9-8	20-7-6-5-4-3-2-1
22	110	109	110	21-20-19-18-17-16-15-14-13-12-11-10-1	22-9-8-7-6-5-4-3-2
24	119	118	119	23-22-21-20-19-18-17-16-15-14-13-12-10	24-11-9-8-7-6-5-4-3-2-1
26	125	126	126	25-24-23-22-21-20-19-18-17-16-15-14-13-11	26-12-10-9-8-7-6-5-4-3-2-1
28	129	129	129	27-26-25-24-23-22-21-20-19-18-17-16-15-14-1	28-13-12-11-10-9-8-7-6-5-4-3-2
30	141	141	141	29-28-27-26-25-24-23-22-21-20-19-18-17-16-15-14-1	30-13-12-11-10-9-8-7-6-5-4-3-2

Y ahora representaremos las soluciones obtenidas cuando disponemos de 3 líneas:

Tamaño	**T L1**	**T L2**	**T L3**	**T Sol**	**Línea 1**	**Línea 2**	**Línea 3**
2	4	0	7	7	1	-	2
4	18	11	16	18	3	2-1	4
6	30	29	16	30	5-4	3-2-1	6
8	32	32	31	32	7-5-2-1	6-4	8-3
10	35	36	36	36	9-7-5-1	8-6-2	10-4-3
12	39	39	40	40	11-10-9-8-7-1	6-4-2	12-5-3
14	50	50	49	50	13-12-11-10-9-6	8-7-5-4	14-3-2-1
16	52	54	54	54	15-14-13-12-11-10-1	9-7-6-5-2	16-8-4-3
18	56	56	56	56	17-16-15-14-13-12-11-10	9-7-6-4-2	18-8-5-3-1
20	66	66	66	66	19-18-17-16-15-14-13-11	12-10-9-7-6-4-2	20-8-5-3-1

3 Problema: TSP

3.1 Descripción de la solución

Como ya hemos visto en prácticas anteriores, el problema del viajante de comercio trata de encontrar la manera más óptima de realizar un recorrido entre diferentes ciudades, volviendo finalmente a la ciudad de la que se parte.

En este caso trataremos de dar solución al problema utilizando el método de ramificación y poda. Para ello lo que hacemos es comprobar si quedan ciudades por visitar. En el caso de que no queden ciudades por visitar (estamos en un nodo hoja), se calcula la distancia total del camino que hemos hecho (sumando finalmente la distancia desde la última ciudad visitada hasta la ciudad de la que partimos para cerrar el ciclo). En el caso de que la distancia que acabamos de obtener sea menor que la mejor solución obtenida hasta ese momento, nos quedaremos con la última distancia calculada como la mejor. Por otro lado, si quedasen ciudades por visitar, procederíamos a ordenar la cola de prioridad. A continuación, para cada ciudad que nos falte por visitar calcularemos la distancia del camino al añadir la ciudad i-ésima no visitada y la distancia más optimista estimada del resto de ciudades. Si la suma de las dos distancias calculadas anteriormente es menor que la solución real, entonces procedemos a expandir ese nodo y a resolver (ejecutar la función recursivamente). En caso contrario podaríamos la rama. Cabe destacar que en nuestra implementación la lista de nodos vivos (lista de ciudades sin visitar) inicialmente siempre tendrá de tamaño n-1 (siendo n el número total de ciudades a recorrer), ya que la primera ciudad siempre es fija y no la contaremos hasta que visitemos las demás.

3.2 Pseudocódigo

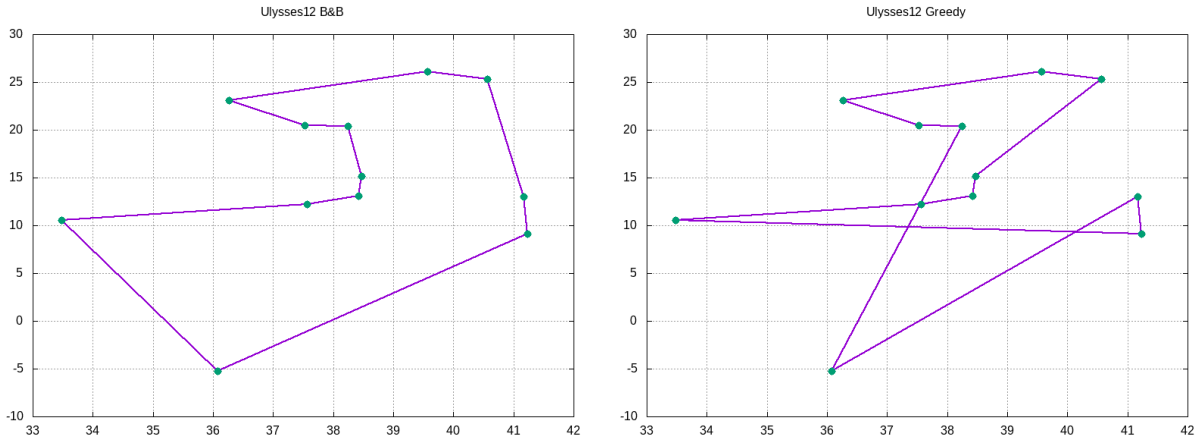
```
csv:=vector de ciudades sin visitar
cv:=vector de ciudades visitadas
dv:=distancia ciudades visitadas
de:= distancia estimada
ds:= distancia solucion
ne:= nodos explorados
n_podas := numero de podas realizadas
cs:= camino solucion

funcion resolver(cv, csv, dv, de)
cv_aux, csv_aux := vector    //vectores auxiliares de ciudades visitadas y sin
   visitar
dt, de_aux := int           //dt=distancia total

inicio
    if(!cv.vacio)
        //ordenar por prioridad las ciudades sin visitar
        csv := prioridad(cv,back(), csv)
        cv_aux := cv

        desde i := 0 hasta i < csv.size()
            dt := 0
            cv_aux := cv
            //Se introduce la primera ciudad del vector sin visitar ordenado
               por prioridad
            cv_aux.push_back(csv[i])
            csv_aux := csv
            //Se elimina la ciudad del vector de ciudades sin visitar auxiliar
            csv_aux.erase(csv_aux.begin()+i)
            //Se incluye la distancia de la ciudad a la distancia total
            dt += dv + distancias[cv.back()][cv_aux.back()]
            //Se calcula la nueva distancia estimada
            de_aux = de - menorArista(cv_aux.back())
```

```
            if (dt + de_aux < ds)
                ne++                //incremento nodos explorados
                resolver(cv_aux, csv_aux, dt, de_aux)    //entrada recursiva
            fi
            else n_podas++      //incremento nodos podados
        fin
    fi

    else    //No quedan valores en ciudades visitadas para analizar
        dt := 0
        desde i:=0 hasta cv.size()-1
            dt += distancias[cv[i]][cv[i+1]]
        fin

        dt += distancias[cv.front()][cv.back()]

        if (dt < ds) // si distancia total menor que distancia solucion
            ds = dt  // Sustituye distancia solucion por distancia total
            cs = cv  // Sustituye camino solucion por ciudades visitadas
        fi
fin
```

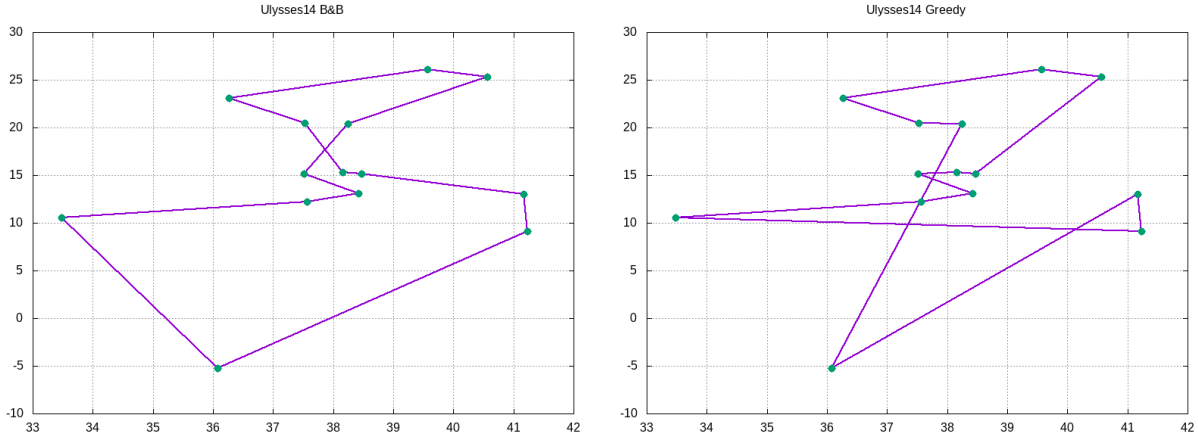
3.3 Escenarios de ejecución

Para la realización de los escenarios, dada la complejidad del problema se ha optado en realizar ejecuciones con el conjunto de datos de Ulysses16, en su manera completa, con 16 ciudades, así como versiones reducidas con menores cantidades de ciudades.

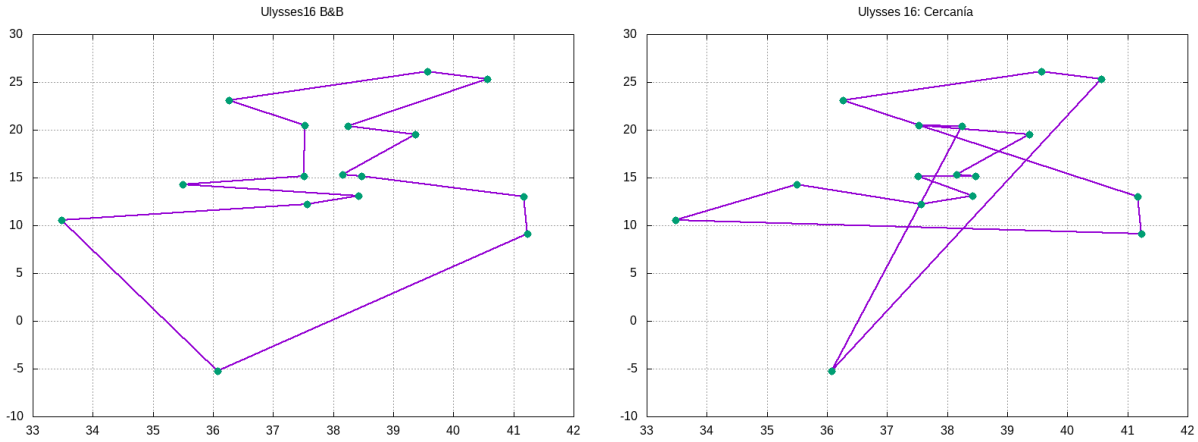
En primer lugar, comparemos lo que obtiene Greedy con lo que se obtiene con B&B:



Ulysses12: B&B vs Greedy

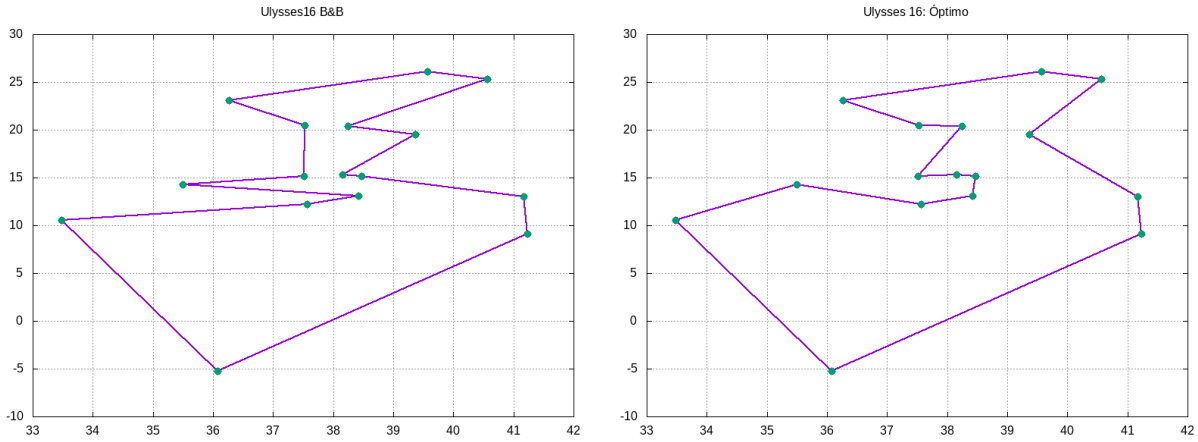


Ulysses14: B&B vs Greedy



Ulysses16: B&B vs Greedy Cercanía

Ahora bien, comparemos Branch and Bound con la ruta provista como óptima de la librería TSPLib:



Ulysses16: B&B vs Óptimo TSPLib

TSP	Greedy	B&B	Tiempo (s)	Nodos Expandidos	Número de Podas	Tamaño máximo de LNV
Ulysses6	36	35	0.000407	166	82	5
Ulysses8	40	36	0.009283	2246	2669	7
Ulysses12	83	68	13.0852	3701838	6443600	11
Ulysses14	84	68	49.8863	217594737	443383865	13
Ulysses16	103	71	3848.92	16799129630	39133662533	15

De primera mano se puede observar que el camino obtenido por el algoritmo de Branch and Bound posee la características del óptimo, en el sentido que la ruta traza una especie de contorno sobre las ciudades y rara vez existen cruces en las rutas, y en efecto, la distancia obtenida es substancialmente menor que la que provee el Greedy y concuerda con los valores que se han obtenido en Programación Dinámica, si bien las rutas difieren poseen la misma distancia óptima de 71, esto desmuestra cómo el Branch and Bound es otra técnica útil para resolver problemas de optimización de tipo NP Duro y en este caso que pueden ser representados como árboles implícitos.

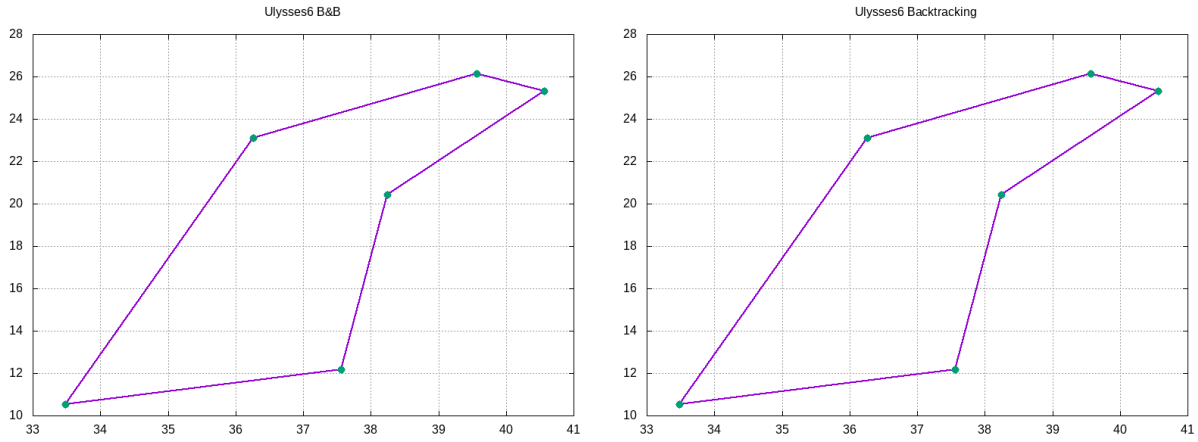
3.4 Comparación entre B&B y Backtracking

Hemos medido los tiempos de ejecución de ambos algoritmos con escenarios de 6, 8, 12, 14 y 16 ciudades (ulysses6, ulysses8, ulysses12, ulysses14 y ulysses16) y hemos obtenido la siguiente tabla:

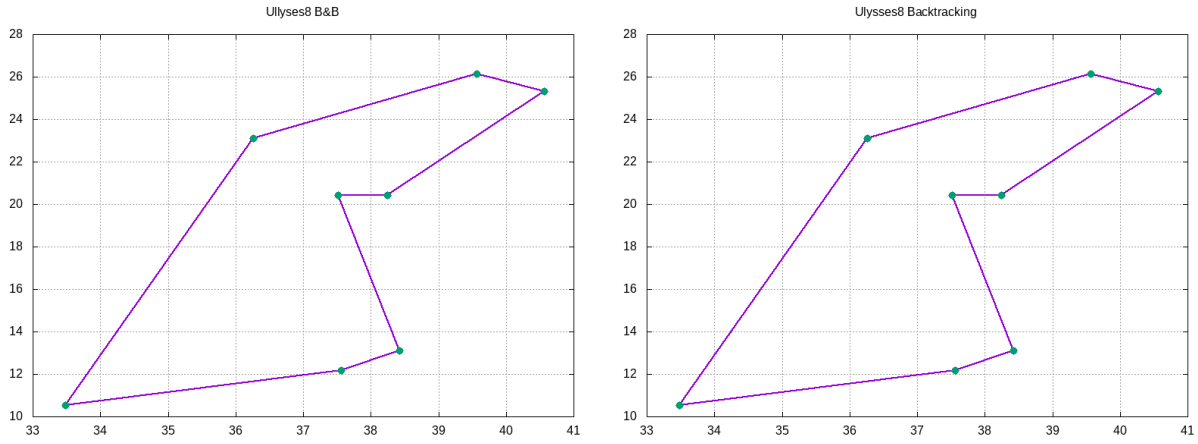
TSP	Tiempo ejecución Backtracking (seg)	Tiempo ejecición B&B (seg)
Ulysses6	3.5e-05	4.2e-05
Ulysses8	0.001272	0.000578
Ulysses12	9.87395	0.750011
Ulysses14	1614.15	46.6722
Ulysses16	338971,5	3861.45

Aquí podemos apreciar que los tiempos de ejecución en el caso de Backtracking son mucho mayores que en el caso de B&B debido a que este último no comprueba todas las posibilidades al podar algunas ramas del árbol binario. El hecho de añadir la poda al algoritmo hace que la eficiencia de B&B sea notablemente menor que la de Backtracking. Cabe destacar que el tiempo de ejecución de ulysses16 con el algoritmo backtracking es un cálculo aproximado, ya que realizarlo de forma experimental nos llevaría días debido a que la eficiencia de este algoritmo de factorial. Para hacer este cálculo aproximado nos hemos basado en que si para las 13! combinaciones que tiene ulysses14 el algoritmo tarda 1614.15 segundos, el mismo algoritmo tardará de forma aproximada unas 120 veces más en hacer las 15! de ulysses16 ya que 15! = 15*14*13!.

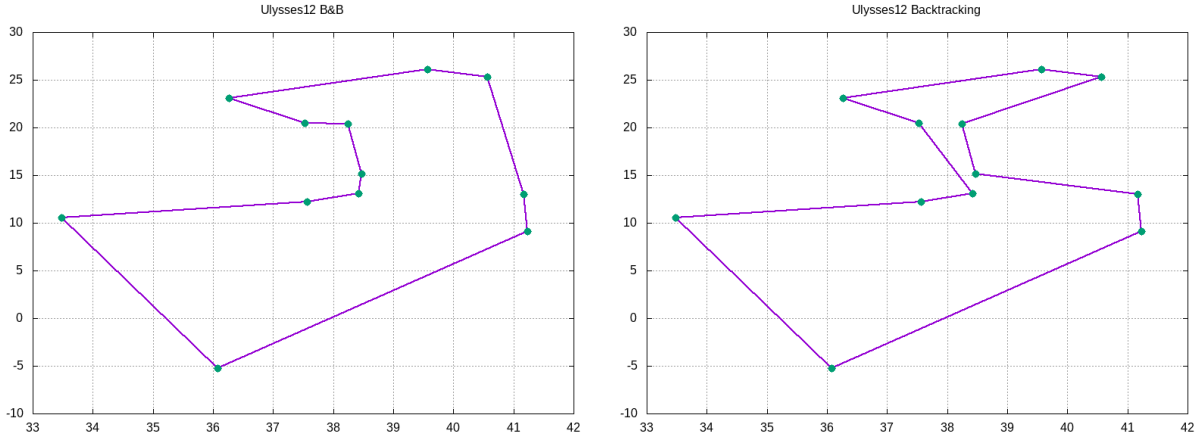
A continuación mostramos los caminos devueltos por ambos algoritmos para los escenarios tratados (excepto para ulysses16, por lo explicado anteriormente):



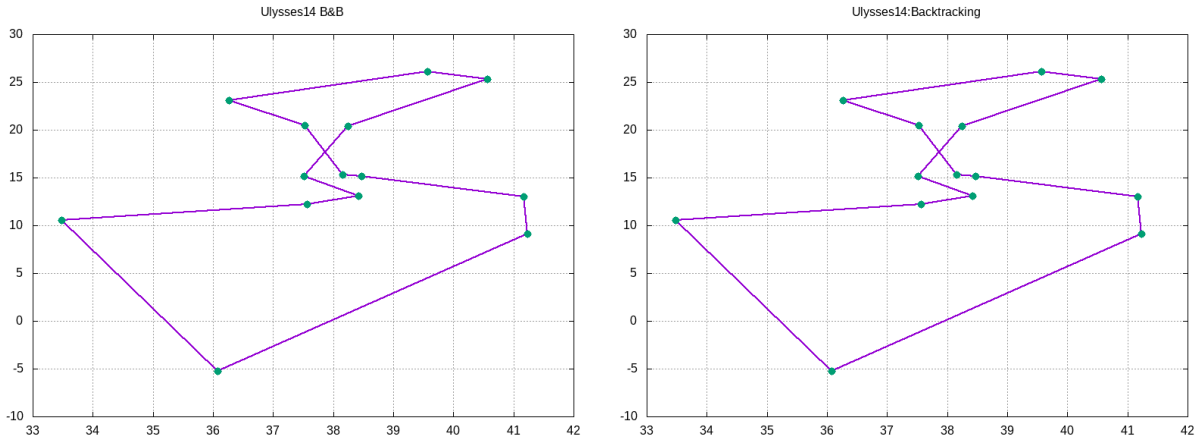
Ulysses6: B&B vs Backtracking



Ulysses8: B&B vs Backtracking



Ulysses12: B&B vs Backtracking



Ulysses14: B&B vs Backtracking

4 Conclusión

De manera similar a la Programación Dinámica, los algoritmos de exploración de árboles dado que la eficiencia puede variar mucho dependiendo de la manera que se implementen, las cotas que se utilicen y los datos mismos, teniendo normalmente un peor caso que varía entre orden exponencial y orden factorial, como se pudo observar en la práctica dado que el problema del Viajante del Comercio es NP completo. Asi mismo, se pudo observar la diferencia entre Branch & Bound y Backtracking, dado que Backtracking en problemas de optimización no es recomendado puesto que va a explorar todas las ramas del árbol implícito, brillando en problemas de satisfacción de condiciones, como por ejemplo, recorrer un laberinto.