Práctica 5
Algoritmos de Vuelta Atrás (Backtracking) y de Ramificación y Poda (Branch and Bound)
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 realizar dos ejercicios.

El primer problema es el de “Cena de gala”, el cual trata de establecer de qué manera se tienen que sentar un número determinado de comensales de forma que la conveniencia total sea máxima, entendiendo la conveniencia como un valor indicador de cuan acertado es sentar un comensal con otro. Esto lo vamos a medir con un valor del rango [0, 100] donde 0 indica una conveniencia pésima y 100 una conveniencia perfecta. Este problema lo vamos a resolver mediante la técnica de Backtracking, el cual se basa en una búsqueda por profundidad.

El segundo problema se trata del “Viajante de comercio”, el cual resolveremos mediante la técnica de Branch and Bound. Este problema trata de averiguar cuál es la ruta óptima entre varias ciudades por la que debe pasar un viajante de forma que la distancia recorrida sea mínima, dicho problema se basa en una búsqueda de costo uniforme.

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 2 GHz.

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

```
$lscpu
Arquitectura:                        x86_64
modo(s) de operación de las CPUs:    32-bit, 64-bit
Orden de los bytes:                  Little Endian
Tamaños de las direcciones:          39 bits physical, 48 bits virtual
CPU(s):                              4
Lista de la(s) CPU(s) en línea:      0-3
Hilo(s) de procesamiento por núcleo: 2
Núcleo(s) por «socket»:              2
«Socket(s)»                          1
Modo(s) NUMA:                        1
ID de fabricante:                    GenuineIntel
Familia de CPU:                      6
Modelo:                              69
Nombre del modelo:                   Intel(R) Core(TM) i7-4510U CPU @ 2.00GHz
```

3. Cena de gala

Para resolver este problema vamos a utilizar la técnica de Backtracking, este trata de realizar un recubrimiento de un árbol por profundidad, donde cada rama genera una solución parcial hasta que llega a un nodo hoja (último comensal en sentarse) donde se establece una nueva solución en base a la anterior si la hubiera (si mejora se actualiza).

3.1 Planteamiento Backtracking

Buscamos sentar a una cantidad N de comensales en una mesa, de tal forma que la conveniencia total generada sea máxima, teniendo en cuenta que cada par de comensales tendrá una conveniencia distinta entre ellos.

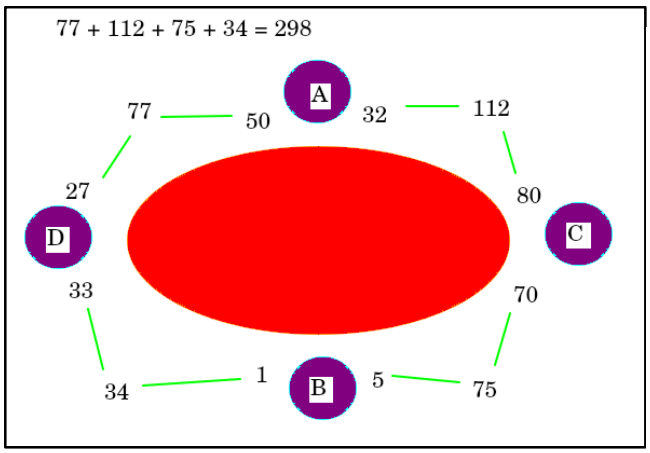
Utilizamos una matriz de conveniencia donde guardamos la conveniencia entre cada par de comensales, evaluada desde 0 a 100. De esta forma, una conveniencia 0 es muy mala y una conveniencia 100 es muy buena (teniendo en cuenta que un comensal no se puede sentar consigo mismo).

Hay que tener en cuenta que entre una persona “A” y otra persona “B”, puede ser que a “A” le caiga muy bien “B” (conveniencia alta) y que a “B” no le caiga tan bien “A” (conveniencia mala), en el siguiente ejemplo de matriz de conveniencia lo vemos:

	A	**B**	**C**	**D**
A	0	10	32	50
B	95	0	5	1
C	80	70	0	60
D	27	33	42	0

El comensal “A” tiene una conveniencia 10 con el comensal “B” mientras que el comensal “B” tiene una conveniencia de 95 con “A”, sumando una conveniencia total entre “A” y “B” de 105 de un total posible de 200 al sentarlos juntos.

En la siguiente imagen podemos ver un ejemplo de la conveniencia al sentar a los comensales en el orden {A, C, B, D, A} donde se obtiene una conveniencia de 298.



CONV. PAR	TOTAL PAR	CONV. TOTAL
{A,C} = 32 {C,A} = 80	112	298
{C,B} = 70 {B,C} = 5	75	
{B,D} = 1 {D,B} = 33	34	
{D,A} = 27 {A,D} = 50	77	

Nótese que sentar a los comensales en el orden {A, C, B, D, A} es lo mismo que sentarlos como {C, B, D, A, C}, {B, D, A, C, B} y {D, A, C, B, D}, por lo que, al buscar el orden, da igual por cual comensal se empiece a sentar.

Para representar un comensal hemos usado un struct donde almacenamos el identificador del comensal y la conveniencia total acumulada hasta dicho comensal al sentarlo, de esta forma, podemos saber si la conveniencia total que puede obtener esa rama tiene opción a mejorar la solución anterior si la hay, ya que si quedan dos comensales por sentar y los dos comensales sentados llevan una conveniencia parcial total de 96, la solución optimista de esa rama seria 96 + 200*2 = 496 (el máximo de conveniencia entre dos comensales seria 200), así, si la solución previa nos dio 600 de conveniencia, sabemos que esta rama no nos llevará a una solución mejor y la abandonamos para continuar con otra anterior, ya que 496 < 600.

Así, tras sentar al primer comensal, se generan recursivamente todas las posibles combinaciones para los comensales restantes mientras se va calculando la conveniencia acumulada para cada comensal sentado, de forma que en si en algún momento, la conveniencia total restante, que de forma optimista pueda ser sumada, no supera la conveniencia de la solución anterior dejará de probar combinaciones, volveremos a la bifurcación (nodo padre) y probando con otro comensal (Backtracking).

Por este motivo, se utiliza este método de factibilidad ya que si llegáramos a cada nodo hoja que genere el problema, tendríamos un total de n! permutaciones.

Para devolver la solución usamos un vector de longitud N (número total de comensales), donde cada posición guardará el struct del comensal sentado en esa posición.

3.2 Restricciones explícitas

El número de comensales es positivo, desde 1 hasta N, donde N es el número total de comensales.

El valor de conveniencia de un comensal con otro está en el rango [0, 100]

3.3 Restricciones implícitas

Un comensal sólo puede estar sentado en un único sitio y un sitio sólo puede estar ocupado por un comensal.

Si una solución parcial optimista no supera la solución final actual, no se puede alcanzar a mejorar dicha solución.

3.4 Explicación paso a paso

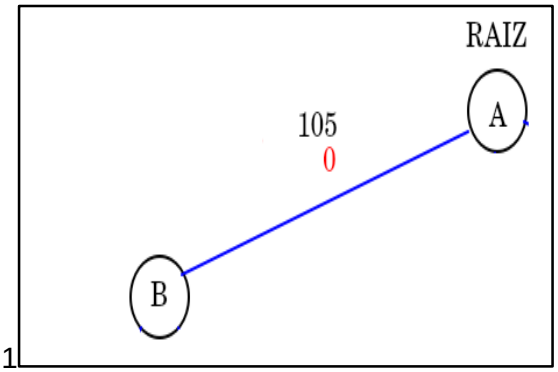
A continuación, vamos a mostrar dos ejemplos de tamaño 4 de ejecución en los que vamos a ver cómo se calcula la conveniencia acumulada y cómo obtiene la solución.

Dada la siguiente matriz de conveniencia entre comensales donde 0 indica muy mala conveniencia y 100 la mejor posible:

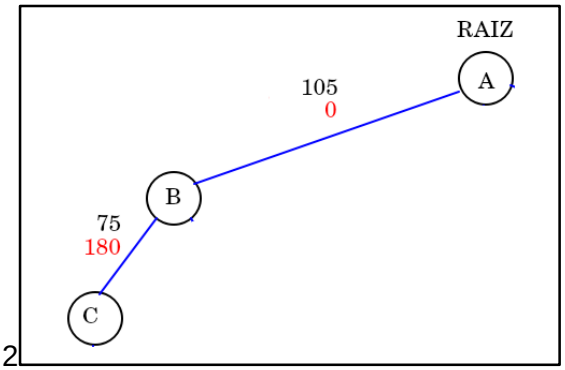
	A	**B**	**C**	**D**
A	0	10	32	50
B	95	0	5	1
C	80	70	0	60
D	27	33	42	0

Nuestro algoritmo iría generando el siguiente árbol paso a paso, priorizando los hijos del nodo generado en vez de generar los nodos hermanos, es decir, en vez de generar {A, B} y luego {A, C}, genera el hijo de {A, B}.

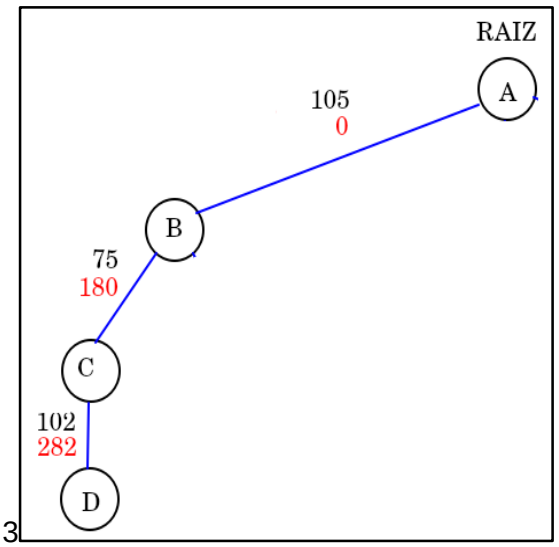
1



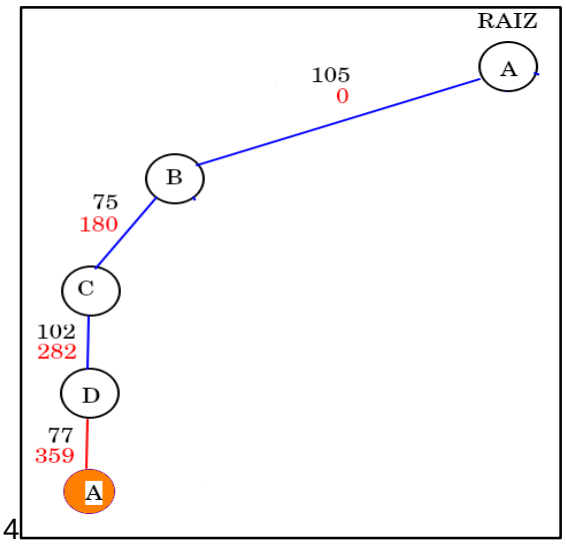
2



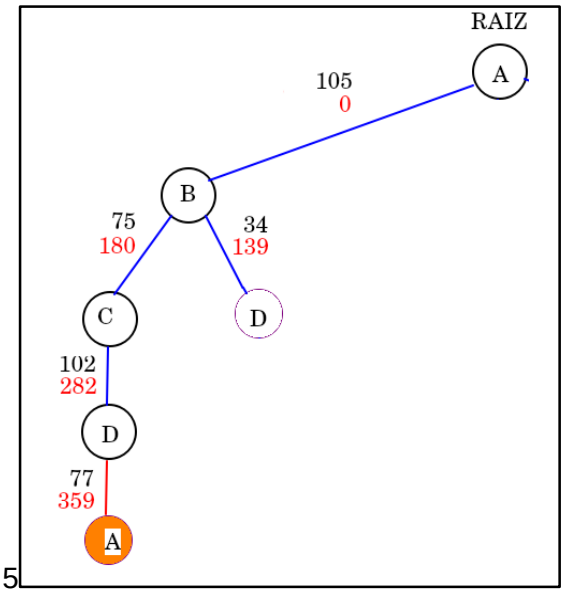
3



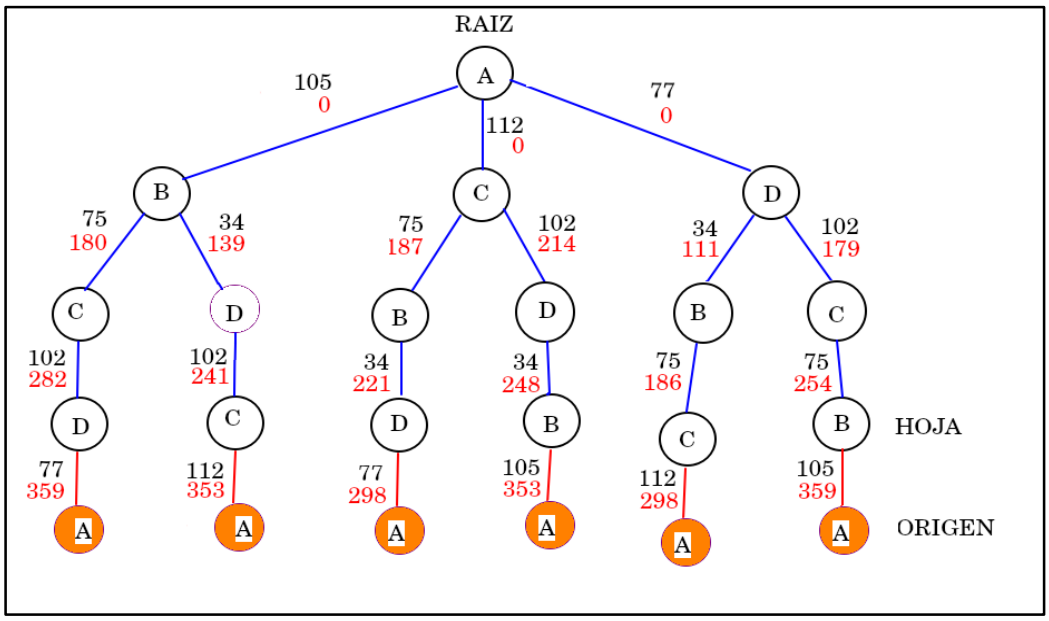
4



5



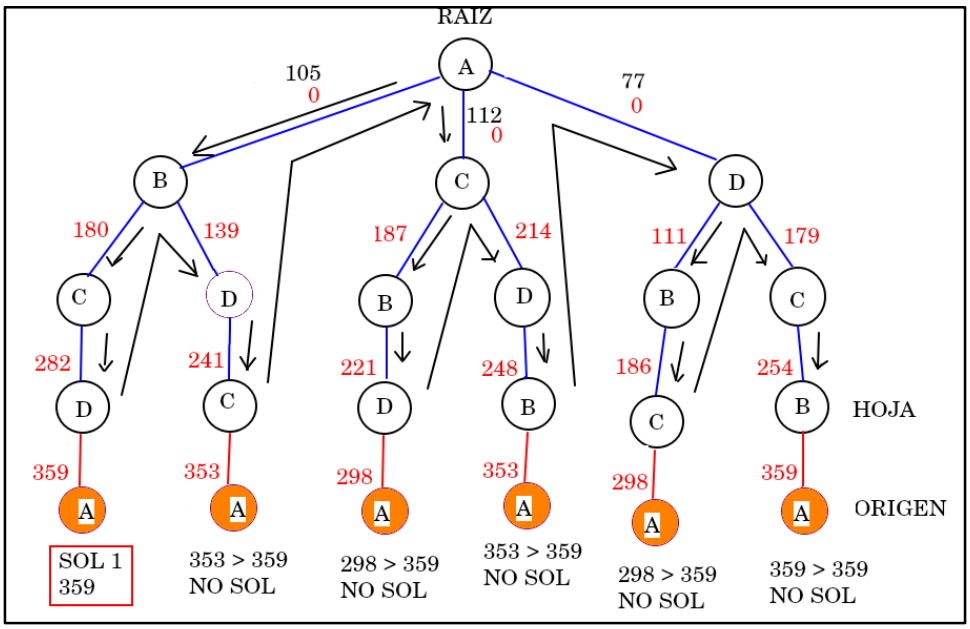
Hasta generar el siguiente árbol:



En ninguno de los nodos hijos generados, la posible conveniencia optimista ha sido menor que la primera conveniencia solución obtenida (359), por lo que el algoritmo pasa por todas las ramas y llega hasta todas las hojas.

A continuación, se puede observar el recorrido del algoritmo Backtracking, cuando termina de evaluar un nodo, pasa a evaluar sus hijos, si el nodo es hoja, añade además la vuelta al origen y termina la recursividad, pasando a evaluar al siguiente nodo hijo de la anterior bifurcación.

En la siguiente imagen podemos ver el recorrido Backtracking que realiza el algoritmo:



Así, podemos ver que obtiene una primera solución en la rama más a la izquierda con recorrido {A, B, C, D, A} con una conveniencia de 359, la cual, no es nunca superada por ninguna otra posible solución al llegar a un nodo hoja.

Se puede ver que la última rama tiene la misma conveniencia, pero no se toma como solución; realmente esto no es así, ya que es la misma solución, aunque vista al revés.

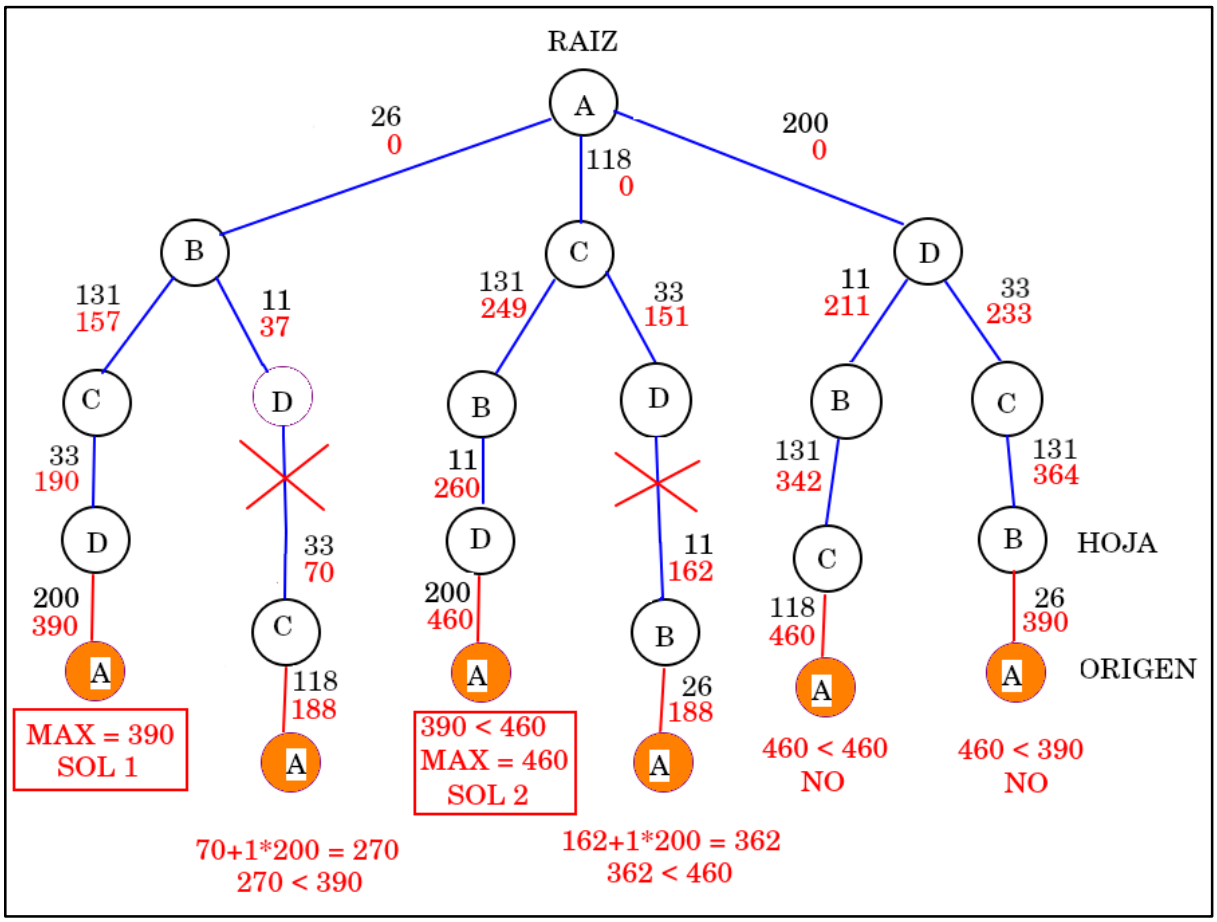
A continuación, se puede ver el resultado obtenido por nuestro programa, donde se confirma la solución obtenida en el árbol visto anteriormente, donde A=1, B=2, C=3, D=4.

```
nonsatus@Non [02/06/20 16:34:43 martes]:~/Documents/UGR1920/ALG/Practicas/P5_BB
$./bin/cena data/cena/cena4.dat 4
Solucion: 1 2 3 4 1
Conveniencia: 359
Tiempo: 2.6539e-05
```

Para el siguiente ejemplo, donde se verá mejor el recorrido de Backtracking, se ha usado la siguiente matriz de conveniencia:

	A	**B**	**C**	**D**
A	0	26	59	100
B	0	0	99	5
C	59	32	0	5
D	100	6	28	0

Que nos generará el siguiente árbol con las conveniencias acumuladas:



En la imagen se pueden ver dos equis rojas, las cuales indican en qué momento se deja de evaluar dicha rama, esto se controla justo antes de llamar al nodo hijo, comprobando si es posible mejorar el resultado con la conveniencia acumulada y la conveniencia más optimista por añadir.

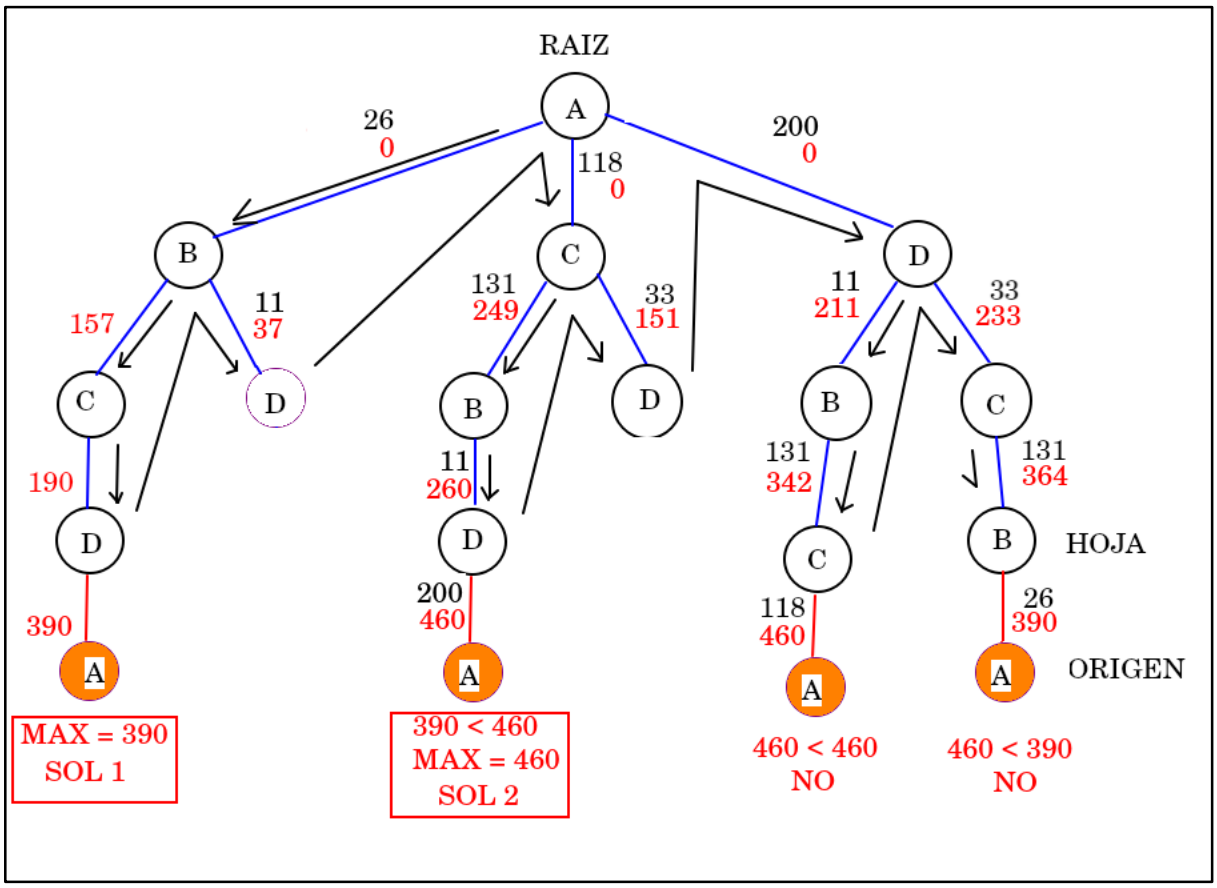
Para ello, primero obtenemos la conveniencia entre padre e hijo recíproca ({D, C} + {C, D} = 33), que sumado a la conveniencia acumulada anteriormente daría un total de 70.

Queda por añadir un comensal en la mesa, por lo que la conveniencia más optimista posible a obtener sería: {C, A}, {A, C} con un valor de conveniencia 200.

Así, la conveniencia más optimista cuando se está en el nodo D, es: 70 + 200 = 270. Y, y 270 es menor que la conveniencia solución anteriormente obtenida, sabemos sin tener que generar el hijo C que no es necesario seguir evaluando dicha rama, por lo que pasará a evaluar el siguiente nodo hijo a generar mediante una vuelta atrás, que en este caso sería {A, C}.

En el caso de que llegara a un nodo hoja, se comprueba si la solución obtenida es la mejor hasta el momento y se actualiza si procede, esta situación se da en la rama {A, C, B, D, A}.

En la siguiente imagen se puede ver el recorrido que haría el algoritmo para este problema:



Vemos que, al ejecutar el programa, se ha obtenido la solución mostrada en el árbol anterior, donde 1=A, 2=B, 3=C y 4=D.

```
nonsatus@Non [02/06/20 16:34:48 martes]:~/Documents/UGR1920/ALG/Practicas/P5_BB
$./bin/cena data/cena/cena4-1.dat 4
Solucion: 1 3 2 4 1
Conveniencia: 460
Tiempo: 2.8634e-05
```

3.5 Eficiencia empírica y resultados

Las siguientes ejecuciones se han realizado ejecutando 5 iteraciones para cada tamaño de problema, donde en cada iteración se genera una matriz aleatoria y esto se realiza debido a que, al aumentar el número de comensales, toda la matriz previa se ve afectada por la conveniencia al nuevo comensal.

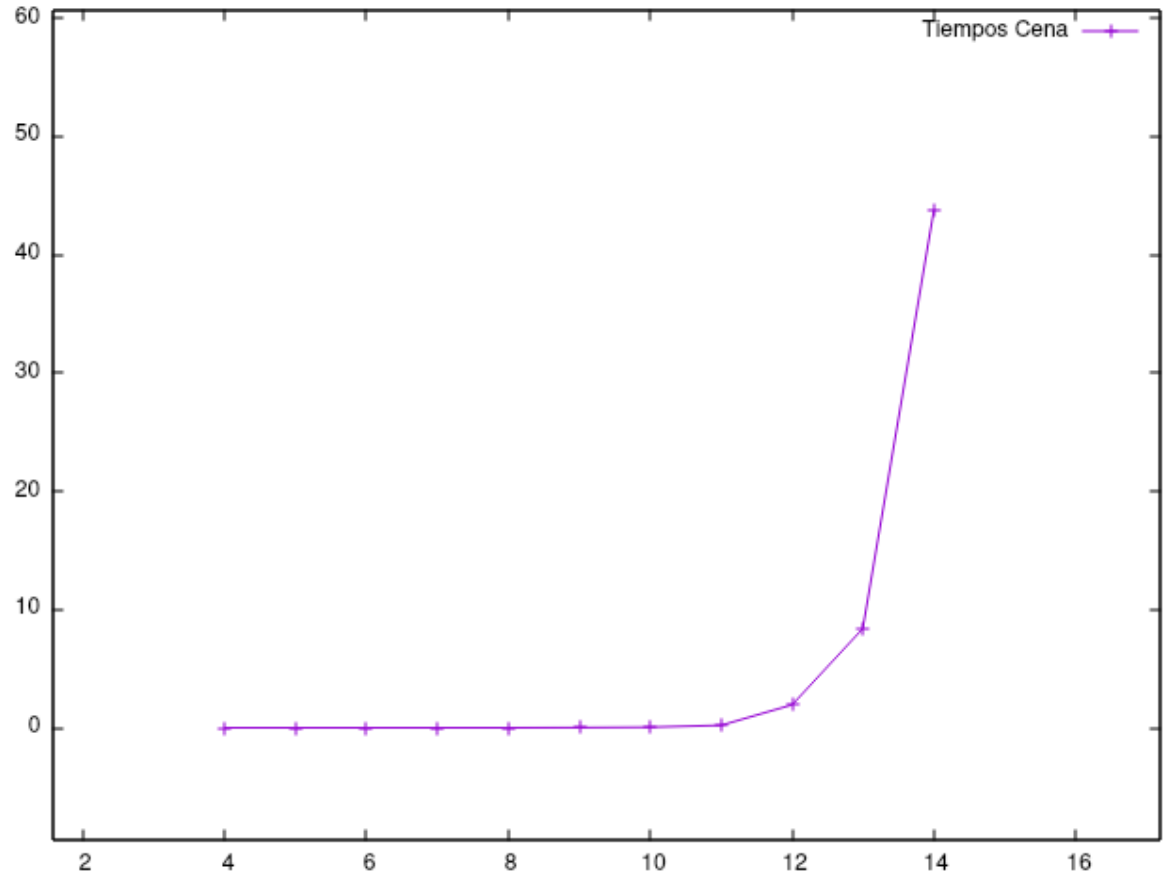
Esto puede producir que la matriz para N=4 siempre llegue a nodos hojas y que para N=5 haga muchas vueltas atrás anticipadas según la función de factibilidad que hemos establecido, por ello obtenemos la media de 5 ejecuciones para cada tamaño de problema.

```
nonsatus@Non [02/06/20 18:02:05 martes]:~/Documents/UGR1920/ALG/Prac
$./bin/cena 4
Comensales: 4
Tiempo: 2.71218e-05
nonsatus@Non [02/06/20 18:02:08 martes]:~/Documents/UGR1920/ALG/Prac
$./bin/cena 5
Comensales: 5
Tiempo: 0.000125995
nonsatus@Non [02/06/20 18:02:09 martes]:~/Documents/UGR1920/ALG/Prac
$./bin/cena 6
Comensales: 6
Tiempo: 0.000391798
nonsatus@Non [02/06/20 18:02:10 martes]:~/Documents/UGR1920/ALG/Prac
$./bin/cena 7
Comensales: 7
Tiempo: 0.00127439
nonsatus@Non [02/06/20 18:02:10 martes]:~/Documents/UGR1920/ALG/Prac
$./bin/cena 8
Comensales: 8
Tiempo: 0.0031716
nonsatus@Non [02/06/20 18:02:12 martes]:~/Documents/UGR1920/ALG/Prac
$./bin/cena 9
Comensales: 9
Tiempo: 0.0131104
nonsatus@Non [02/06/20 18:02:13 martes]:~/Documents/UGR1920/ALG/Prac
$./bin/cena 10
Comensales: 10
Tiempo: 0.0478727
nonsatus@Non [02/06/20 18:02:16 martes]:~/Documents/UGR1920/ALG/Prac
$./bin/cena 11
Comensales: 11
Tiempo: 0.204135
nonsatus@Non [02/06/20 18:02:18 martes]:~/Documents/UGR1920/ALG/Prac
$./bin/cena 12
Comensales: 12
Tiempo: 1.93817
nonsatus@Non [02/06/20 18:02:37 martes]:~/Documents/UGR1920/ALG/Prac
$./bin/cena 13
Comensales: 13
Tiempo: 8.40451
nonsatus@Non [02/06/20 18:03:27 martes]:~/Documents/UGR1920/ALG/Prac
$./bin/cena 14
Comensales: 14
Tiempo: 43.7376
```

A continuación, vemos una tabla de tiempos en segundos para cada tamaño de problema:

N	Tiempo	N	Tiempo
4	0.0000271218	10	0.0478727
5	0.000125995	11	0.204135
6	0.000391798	12	1.93817
7	0.00127439	13	8.40451
8	0.0031716	14	43.7376
9	0.0131104	-	-

Dicha tabla de tiempos nos genera la siguiente gráfica donde podemos observar su eficiencia empírica:

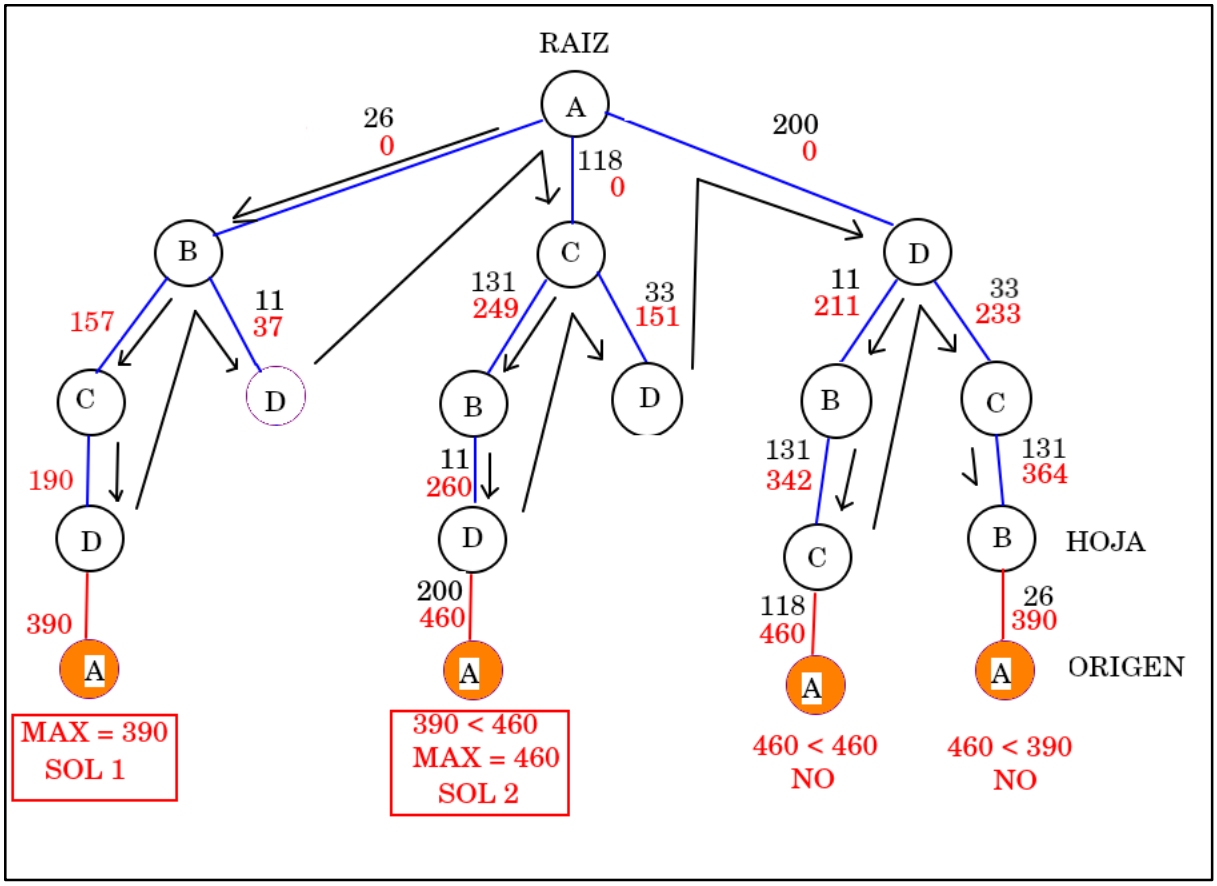


3.6 Pseudocódigo

- Inicializamos los parámetros iniciales:
  - Se crea el nodo raíz con su identificador y la conveniencia total iniciada a 0.
  - Se crea un vector candidatos donde indicamos qué nodos quedan por visitar.
  - Se crea un vector solucion_temporal en el que incluimos el nodo inicial; este vector contendrá la solución propia para cada nodo en todo momento.

- Llamamos a la función recursiva “Backtracking” con los datos iniciales.
  - Marca el comensal candidato
  - Se añade a una solución parcial
  - Si el nodo actual es un nodo hoja:
    - Aumentamos la conveniencia total al añadir este último comensal.
    - Si la solución parcial alcanzada es mejor que la solución previa, actualiza con la nueva solución.

  - Si el nodo actual **no** es un nodo hoja
    - Para cada comensal restante
      - Aumentamos la conveniencia total al añadir al nuevo comensal a la mesa.
      - Si la conveniencia total optimista mejora la conveniencia total de la solución previa
        - Llamada recursiva tomando como válido el comensal evaluado.



4. **Viajante de comercio**

Este problema consiste en obtener una ruta optima por la que debe pasar un viajante entre un conjunto de ciudades, se nos pide resolverlo mediante Branch and Bound y, opcionalmente, mediante Backtracking.

a. **Planteamiento Branch and Bound**

Branch and Bound realiza un recubrimiento de un árbol por costo uniforme, donde cada nodo genera todos sus hijos posibles (anchura), siempre y cuando, aún le sea posible llegar a la solución, eligiendo para la siguiente expansión el hijo con más prioridad según el coste.

Cada vez que lleguemos a un nodo hoja habremos encontrado una posible solución, de las cuales nos quedaremos con la mejor.

Buscamos pasar por todas las ciudades de forma que la distancia recorrida total sea la mínima posible, para ello, usamos una matriz de distancias simétrica donde guardamos las distancias entre cada par de ciudades.

Consideramos todos los distintos caminos que podemos tomar como las distintas ramas de un árbol, donde cada nodo guarda su identificador de ciudad, el camino seguido para llegar hasta ella, las ciudades que aún no han sido visitadas y la distancia total recorrida.

Mientras queden nodos por visitar, nuestro programa utiliza en todo momento el nodo más prometedor (cola con prioridad), es decir, el que lleva registrada menos distancia recorrida total y entonces crea sus hijos, exceptuando aquellos a los que ya no les sea posible llegar a la solución, podando su rama y evitando así recorrerla, al igual que si al generar un hijo este llega a ser hoja, comprueba la posible solución.

Esta poda se realiza cada vez que se cree un nodo cuya distancia recorrida, que corresponde a la cota local, sea mayor o igual que la distancia recorrida por la mejor solución encontrada hasta el momento, representada por la cota global.

Cuando hayamos recorrido o podado todos los distintos nodos significará que hemos encontrado la solución óptima.

Para inicializar la cota global, hemos utilizado la distancia obtenida con un algoritmo Greedy, en este caso mediante Inserción, ya que, de esta forma, podemos usar una cota global aproximada desde el principio, permitiendo descartar ramas que no conllevan a solución.

Para este problema hemos usado las siguientes estructuras de datos:

- struct Ciudad → Almacena el identificador de la ciudad y un par de coordenadas
- struct Nodo → Almacena el identificador que representa la ciudad, el coste acumulado hasta llegar a esa ciudad, un vector que almacena el orden en el que se van visitando las ciudades, un vector que almacena las ciudades restantes candidatas.
- Priority_queue → Almacena los nodos (ciudades) que se van generando de forma que la ciudad con más prioridad es aquella cuyo coste total es menor.
- struct Comparador → Sirve para comparar los nodos (ciudades) introducidos en la cola con prioridad y así ordenarlos según su coste acumulado hasta ese momento.

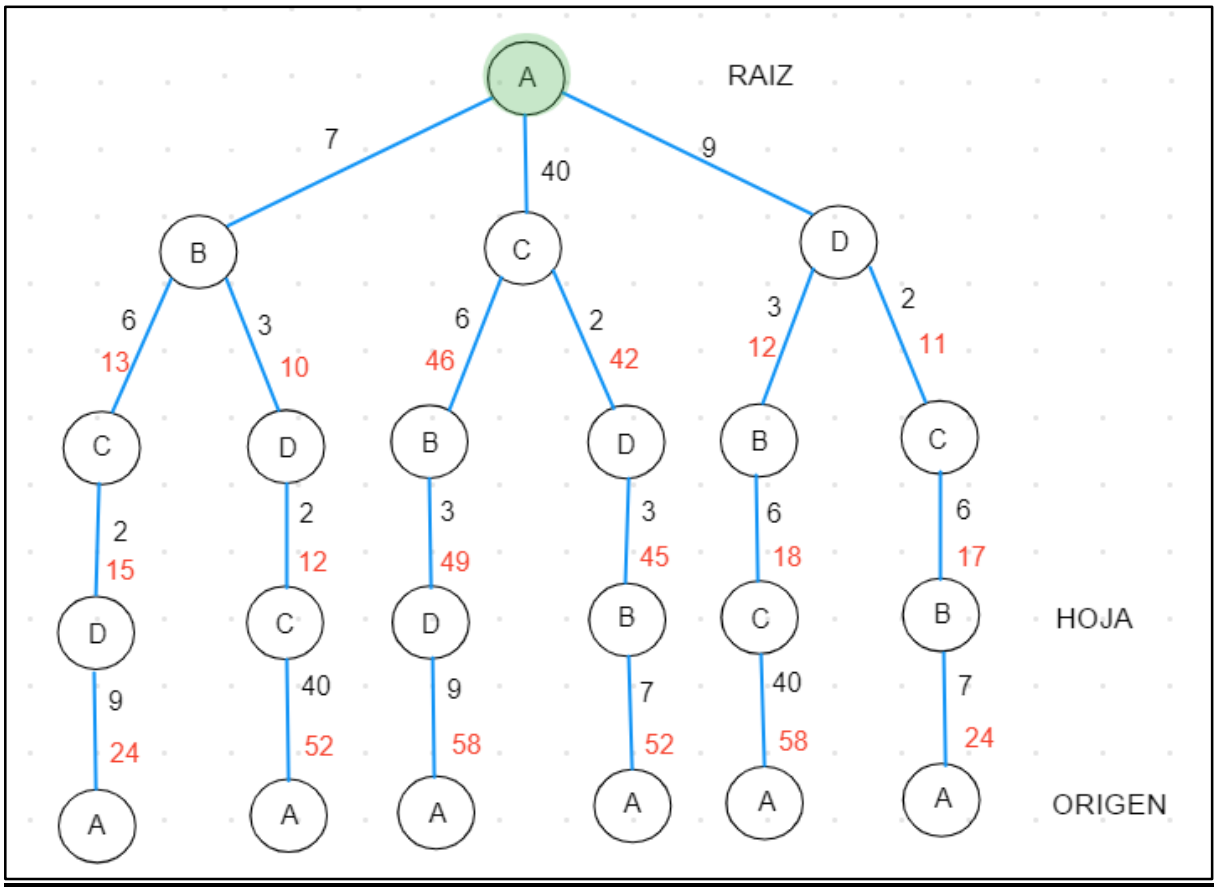
b. **Explicación paso a paso**

Puesto que no es asequible realizar una explicación paso a paso con siete ciudades, ya que el árbol resultante tiene aproximadamente en el peor de los casos (sin podas) unos 720 nodos hojas, procedemos a explicarlo con un ejemplo en el que recorremos 4 ciudades.

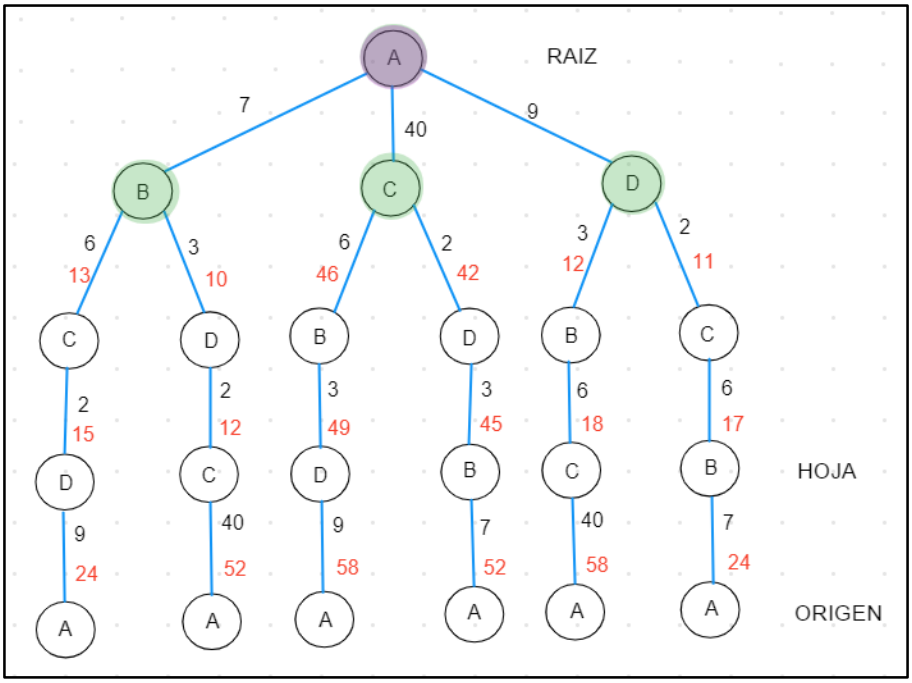
Esta sería la matriz de distancias para dichas ciudades:

	A	B	C	D
A	0	7	40	9
B	7	0	6	3
C	40	6	0	2
D	9	3	2	0

A continuación, vemos la representación de todos los caminos posibles mediante un árbol, donde nuestro conjunto de nodos sin visitar comienza tan solo con el nodo raíz.

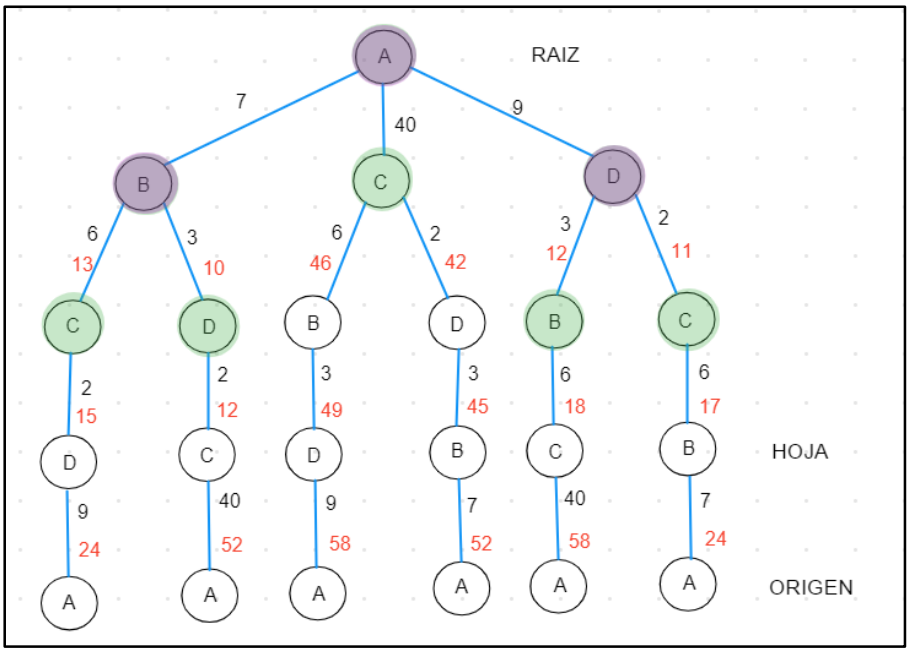


Tomamos el nodo sin visitar con menor distancia, en este caso el nodo raíz y creamos sus tres hijos, añadiéndolos a su vez al conjunto de nodos sin visitar. También eliminamos de este conjunto el nodo ya visitado.

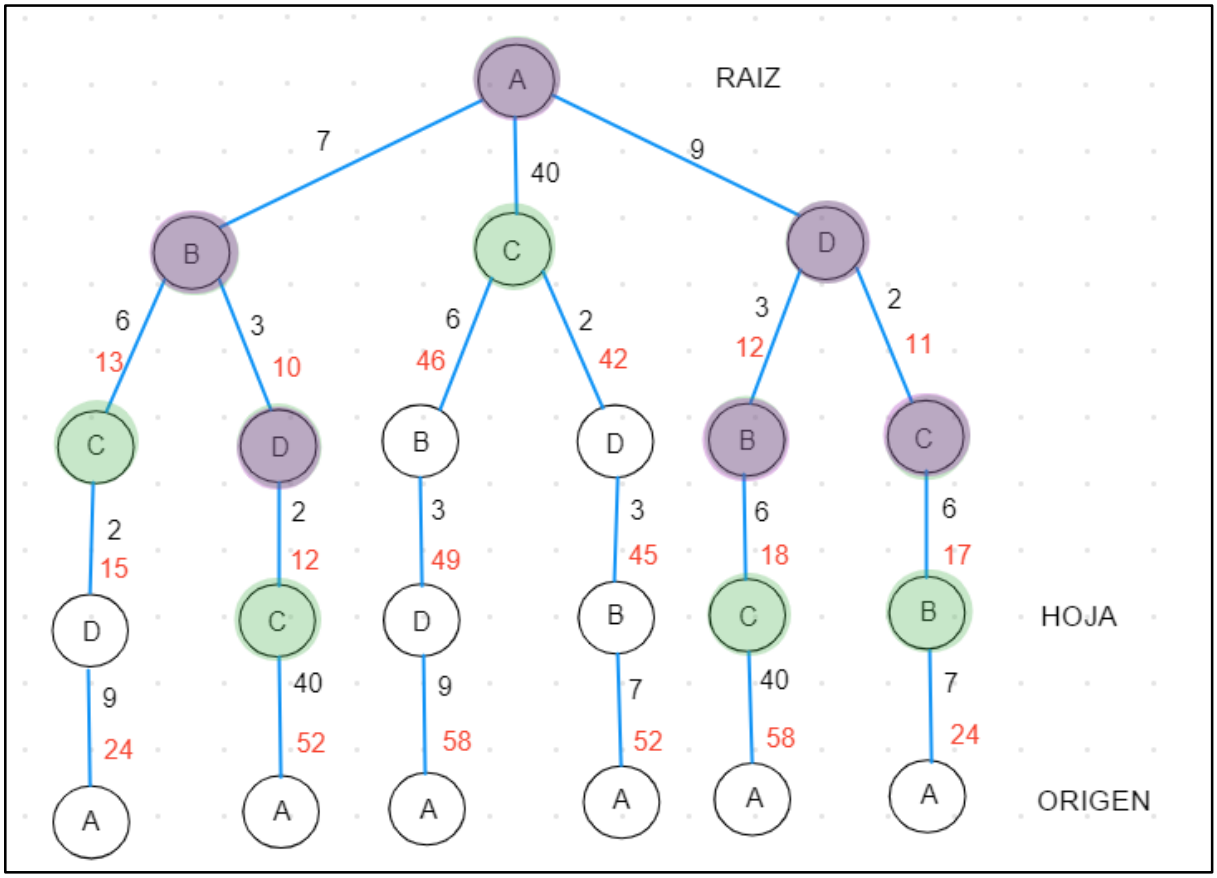


De los tres hijos generados, se elige el mejor nodo según la distancia acumulada, que sería el nodo B con distancia 7, generando sus hijos C y D.

Una vez generado, se escogería el siguiente mejor nodo disponible, que sería el D.



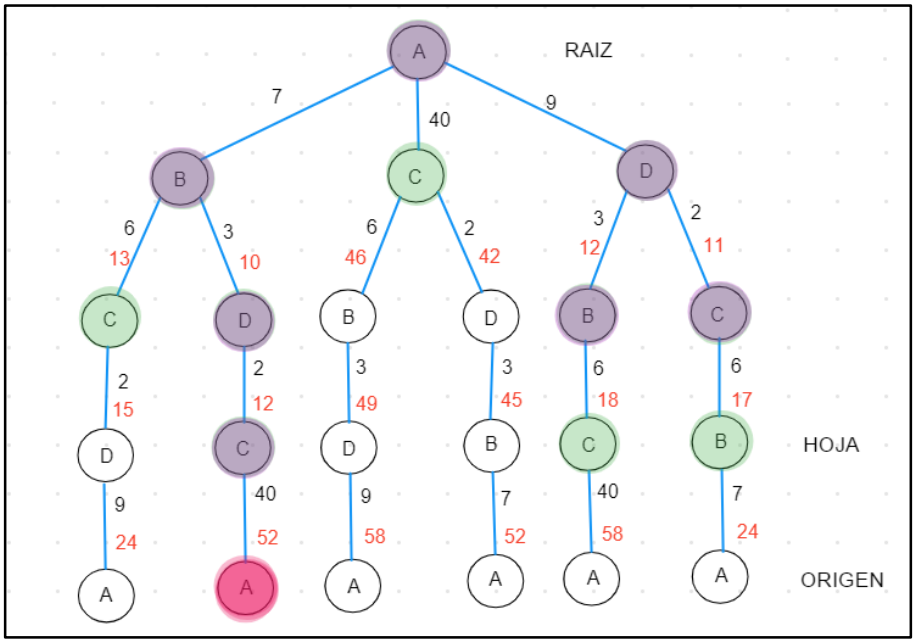
Durante el resto de la ejecución vamos a continuar repitiendo este paso, es decir, tomamos el mejor nodo sin visitar (con menor distancia recorrida), creamos sus hijos y consideramos este nodo como ya visitado.



Estamos creando todos sus hijos porque todavía no hemos calculado ninguna solución válida, pero cuando obtengamos una deberemos comprobar que el hijo pueda llegar a mejorar la solución actual, es decir, si su distancia total todavía es menor que la mejor distancia obtenida.

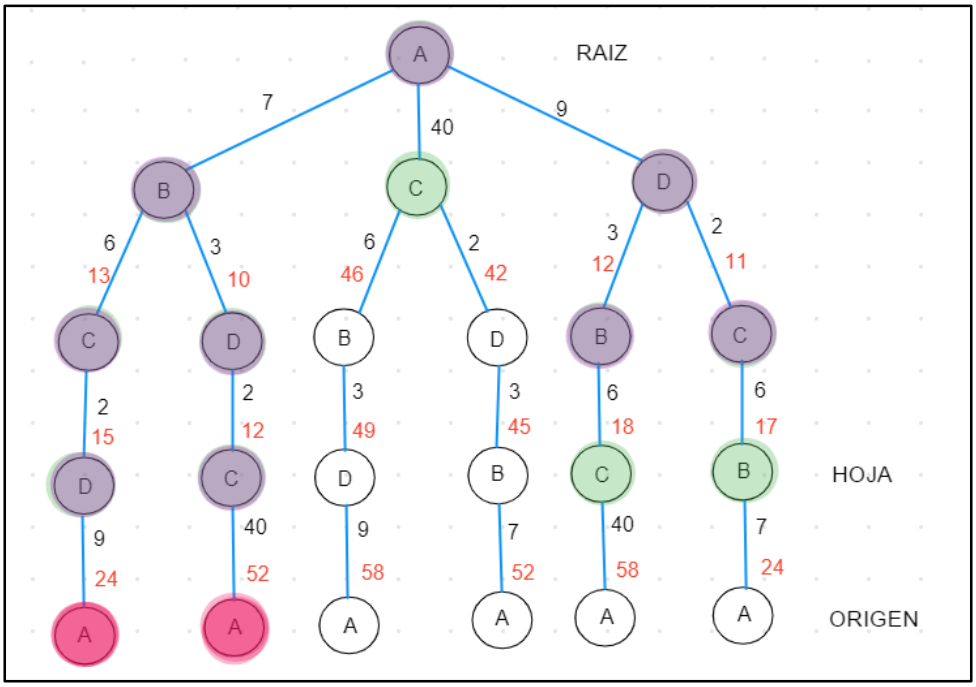
Si no es así, tendremos que podar la rama.

Llegados a este punto nos encontramos que un hijo generado es un nodo hoja, por lo que obtenemos una solución. Como actualmente no tenemos ninguna solución previamente calculada, consideramos {A, B, D, C, A}, con una distancia de 52, como la mejor.



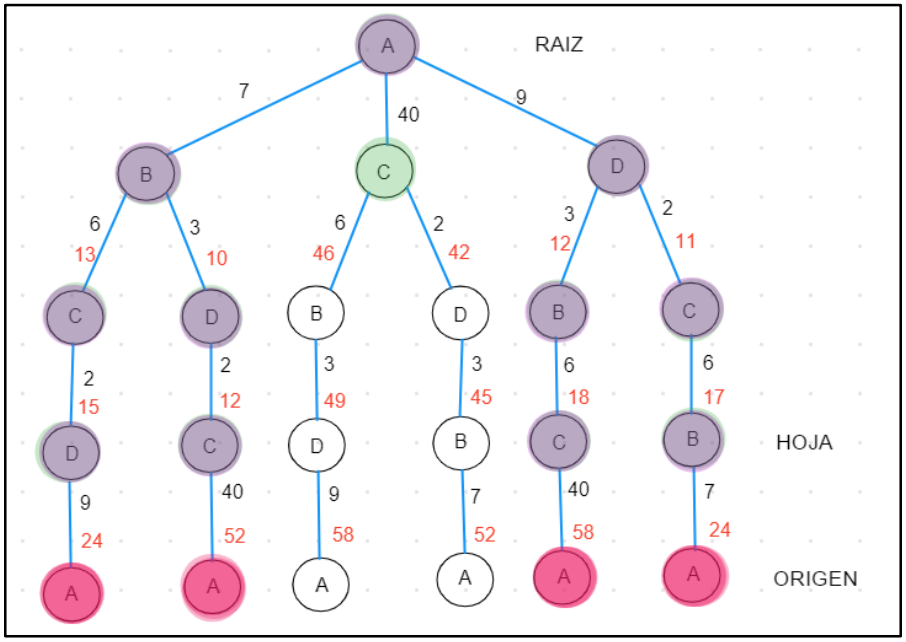
Continuamos con los nodos sin visitar, escogiendo aquellos con menor distancia, en este caso primero con la distancia 13 y luego con la distancia 15; y llegaríamos a otro nodo hoja.

Encontramos otra solución diferente, esta vez con un coste total de 24. Como es mejor que la solución previa, la sustituimos, considerando {A, B, C, D, A} como la mejor solución con una distancia de 24.



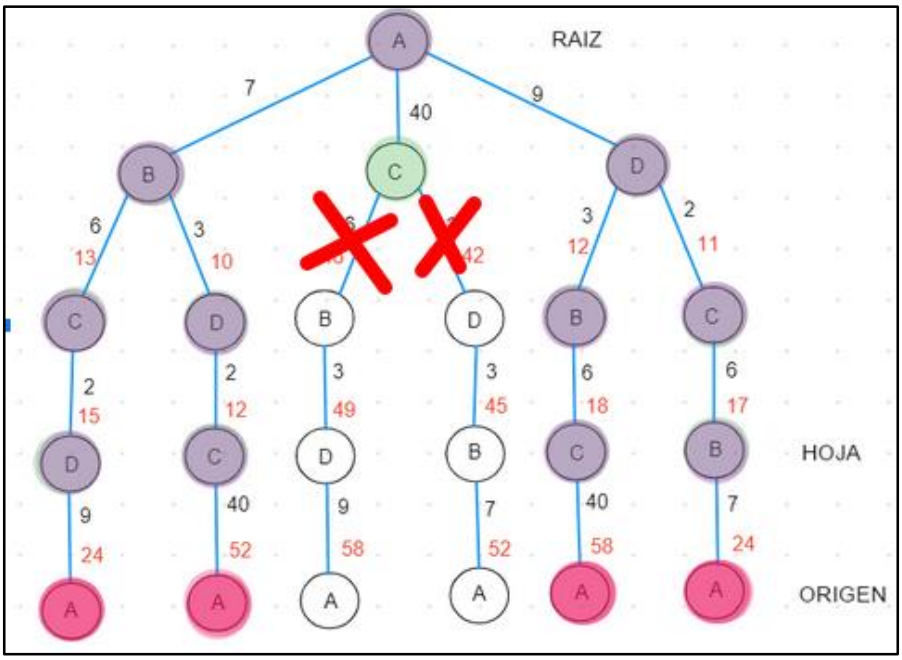
Repetimos el mismo proceso, continuando por los nodos sin visitar con distancia menor.
De esta forma llegaríamos al nodo hoja con distancia 17, encontrando otra solución con coste 24, y seguiríamos por el nodo hoja con distancia 18, encontrando otra solución más con coste 58.

Pero ninguna de ellas es mejor que la solución actual, así que no la modificamos.



Ya solo nos queda un nodo sin visitar, el nodo C con distancia 40.

Este no llega a insertar a sus dos nodos hijos en la cola, ya que estos tienen una distancia de 46 y una distancia de 42, por lo que ninguna de ellas es mejor que la solución actual con coste 24, por lo que podamos esas ramas.



Tras esta poda nos encontramos con que finalmente el conjunto de nodos sin visitar está vacío, ya sea porque los hemos visitado o porque los hemos podado.

Así, damos por finalizada la ejecución y nos quedamos con la mejor solución encontrada, {A, B, C, D, A} con una distancia de 24.

c. **Eficiencia empírica y resultados**

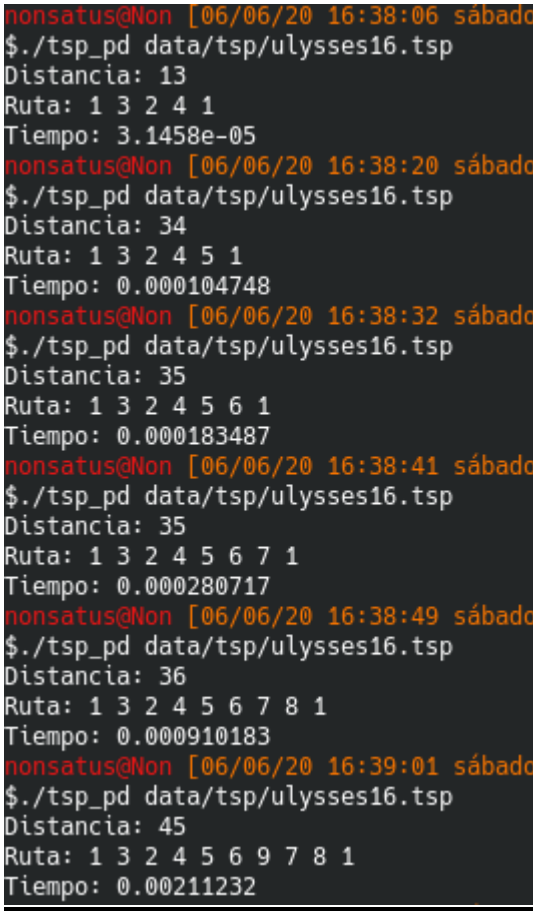
Debido al tipo de algoritmo, Branch and Bound tiene el problema de que tiene un espacio de tamaños limitado, en este caso, para el viajante comercio, lo hemos realizado desde 4 a 12 ciudades, ya que como se ve, para 12 ciudades el tiempo de ejecución crece mucho.

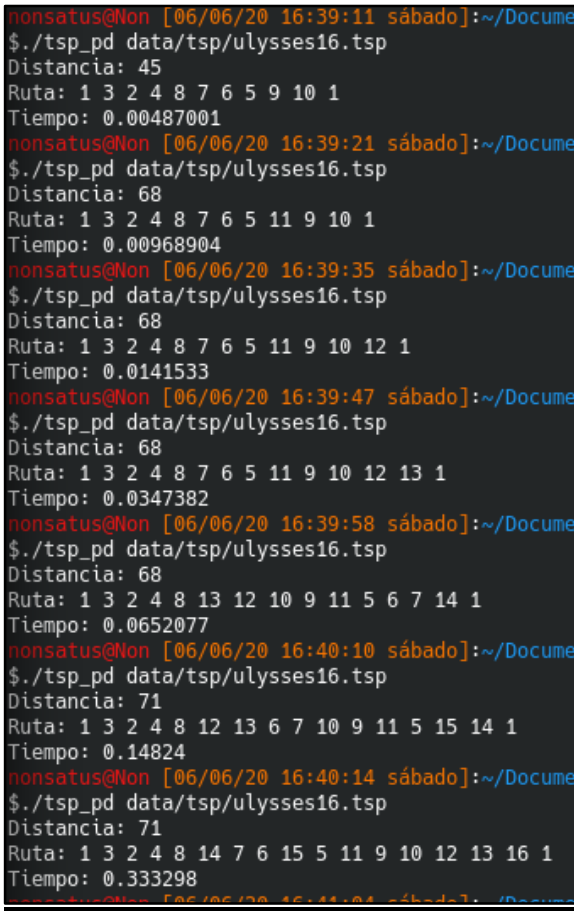
A continuación, se pueden ver las ejecuciones realizadas para el algoritmo B&B.

```
nonsatus@Non [07/06/20 21:55:20 domingo]:
$./bin/tsp data/tsp/ulysses16.tsp
1 3 2 4 1
Distancia: 13
Tiempo: 0.00014634
Nodos expandidos: 10
Capacidad maxima cola: 10
Cantidad de podas: 2
nonsatus@Non [07/06/20 21:55:28 domingo]:
$./bin/tsp data/tsp/ulysses16.tsp
1 3 2 4 5 1
Distancia: 34
Tiempo: 0.000387879
Nodos expandidos: 37
Capacidad maxima cola: 37
Cantidad de podas: 14
nonsatus@Non [07/06/20 21:55:39 domingo]:
$./bin/tsp data/tsp/ulysses16.tsp
1 3 2 4 5 6 1
Distancia: 35
Tiempo: 0.000701301
Nodos expandidos: 137
Capacidad maxima cola: 141
Cantidad de podas: 91
nonsatus@Non [07/06/20 21:55:45 domingo]:
$./bin/tsp data/tsp/ulysses16.tsp
1 3 2 4 5 6 7 1
Distancia: 35
Tiempo: 0.00529736
Nodos expandidos: 467
Capacidad maxima cola: 498
Cantidad de podas: 448
nonsatus@Non [07/06/20 21:55:51 domingo]:
$./bin/tsp data/tsp/ulysses16.tsp
1 3 2 4 8 7 6 5 1
Distancia: 36
Tiempo: 0.0186602
Nodos expandidos: 2112
Capacidad maxima cola: 2128
Cantidad de podas: 2655
```

```
nonsatus@Non [07/06/20 21:55:58 domingo
$./bin/tsp data/tsp/ulysses16.tsp
1 3 2 4 8 7 6 5 9 1
Distancia: 45
Tiempo: 0.0885414
Nodos expandidos: 12335
Capacidad maxima cola: 12335
Cantidad de podas: 17219
nonsatus@Non [07/06/20 21:56:05 domingo
$./bin/tsp data/tsp/ulysses16.tsp
1 3 2 4 8 7 6 5 9 10 1
Distancia: 45
Tiempo: 0.38717
Nodos expandidos: 47001
Capacidad maxima cola: 48915
Cantidad de podas: 81935
nonsatus@Non [07/06/20 21:56:13 domingo
$./bin/tsp data/tsp/ulysses16.tsp
1 3 2 4 8 7 6 5 11 9 10 1
Distancia: 68
Tiempo: 5.33483
Nodos expandidos: 554065
Capacidad maxima cola: 618290
Cantidad de podas: 815572
nonsatus@Non [07/06/20 21:56:25 domingo
$./bin/tsp data/tsp/ulysses16.tsp
1 3 2 4 8 12 7 6 5 11 9 10 1
Distancia: 68
Tiempo: 38.9189
Nodos expandidos: 3785077
Capacidad maxima cola: 4155731
Cantidad de podas: 6206251
```

Para poder compararlo con programación dinámica, hemos ejecutado para el algoritmo PD para 4 ciudades hasta 12 ciudades, de forma que, para cada tamaño de ciudades, podemos comparar los tiempos de ejecución.

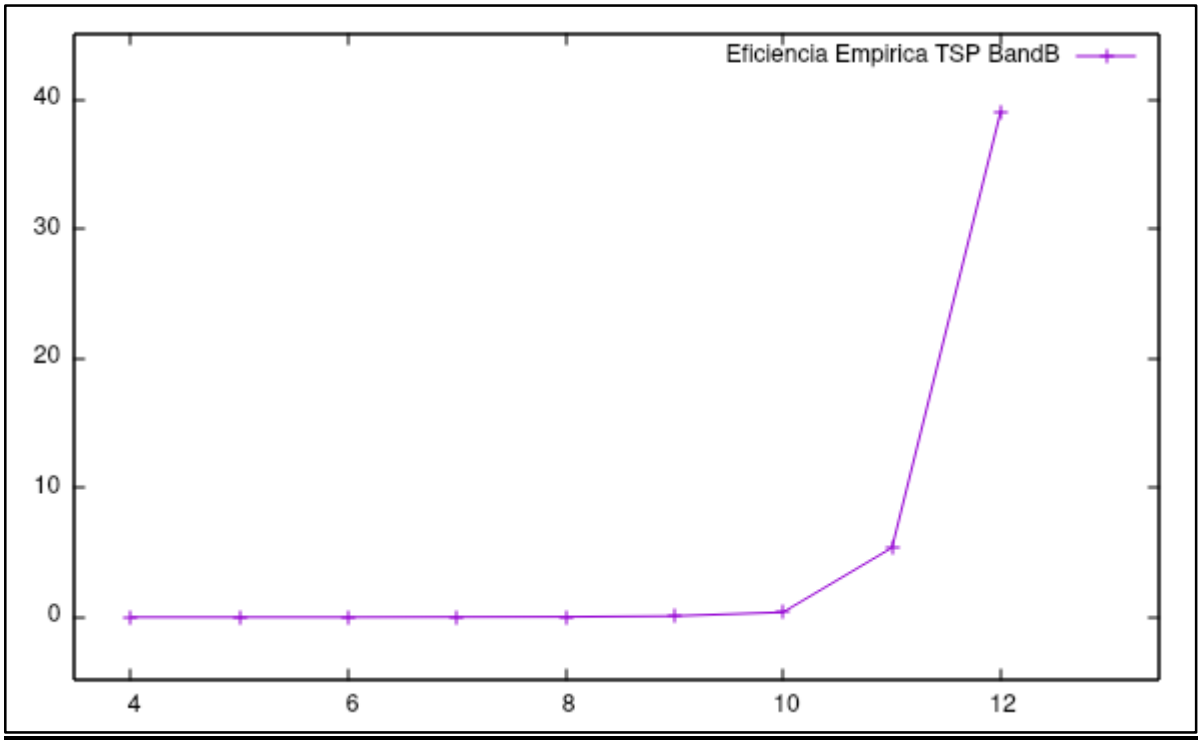




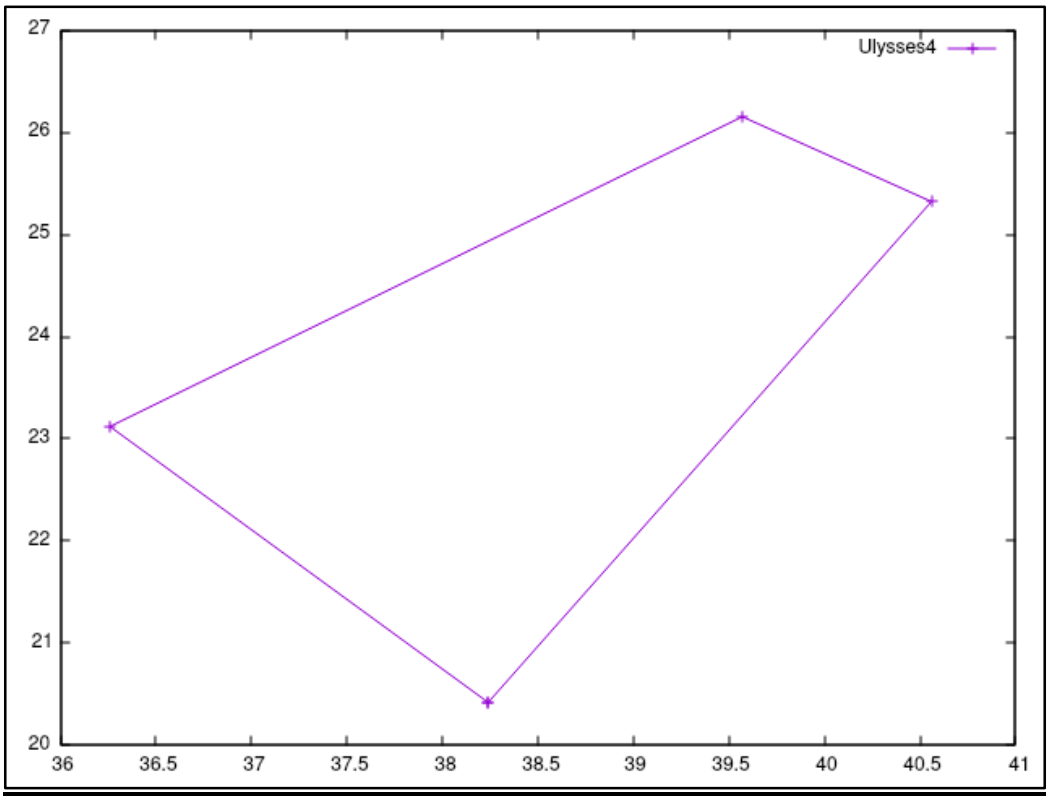
Como se puede ver en las ejecuciones y en la tabla siguiente, el algoritmo de programación dinámica es más rápido que el algoritmo B&B y que ambos obtienen una solución óptima (la misma).

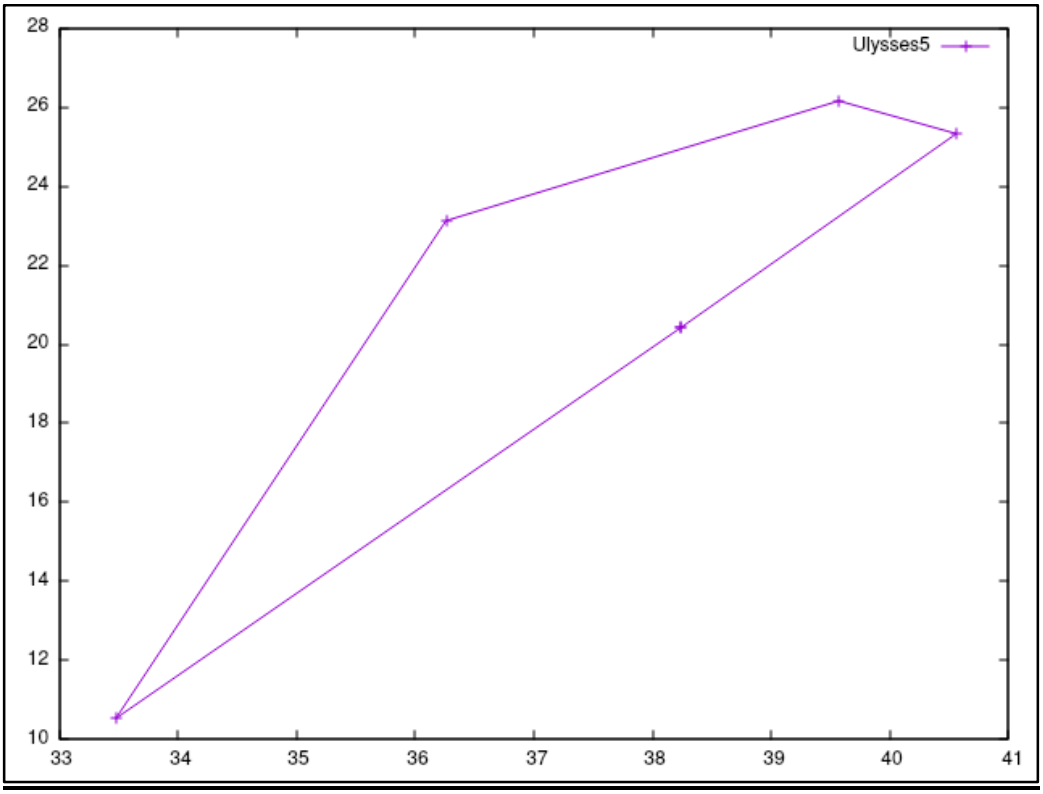
Nº Ciudades	Tiempo (seg)		Nº Ciudades	Tiempo (seg)	
	PD	B&B		PD	B&B
4	0.0000312458	0.00014634	10	0.00487001	0.38717
5	0.000104748	0.000387879	11	0.00968904	5.33483
6	0.000183487	0.000701301	12	0.0141533	38.9189
7	0.000280717	0.00529736	13	0.0347382	-
8	0.000910183	0.0186602	14	0.0652077	-
9	0.00211232	0.0885414	15	0.333298	-

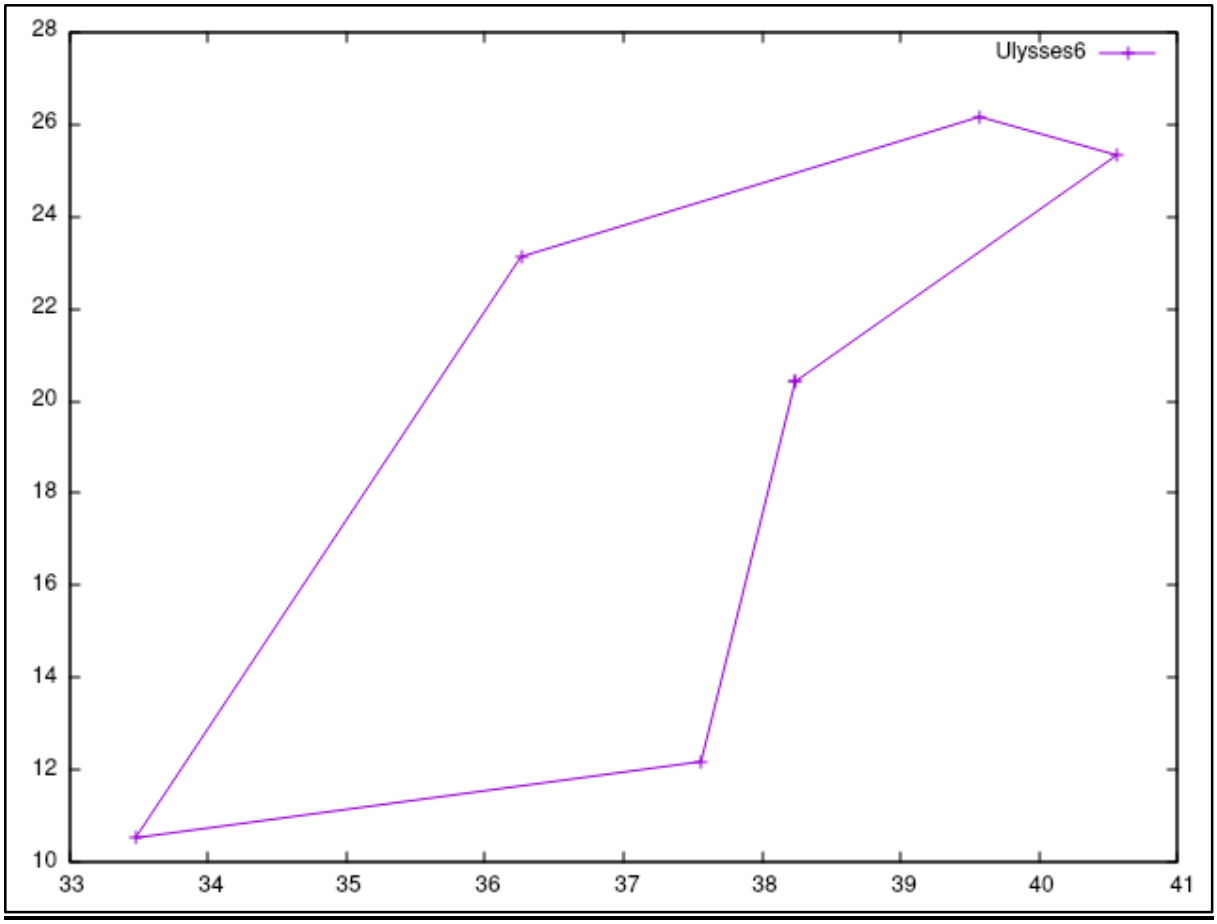
A continuación, podemos ver una gráfica la cual muestra el crecimiento en tiempo según el número de ciudades usadas, viendo que el crecimiento de la curva es muy rápido, lo que limita el número de ciudades a usar.

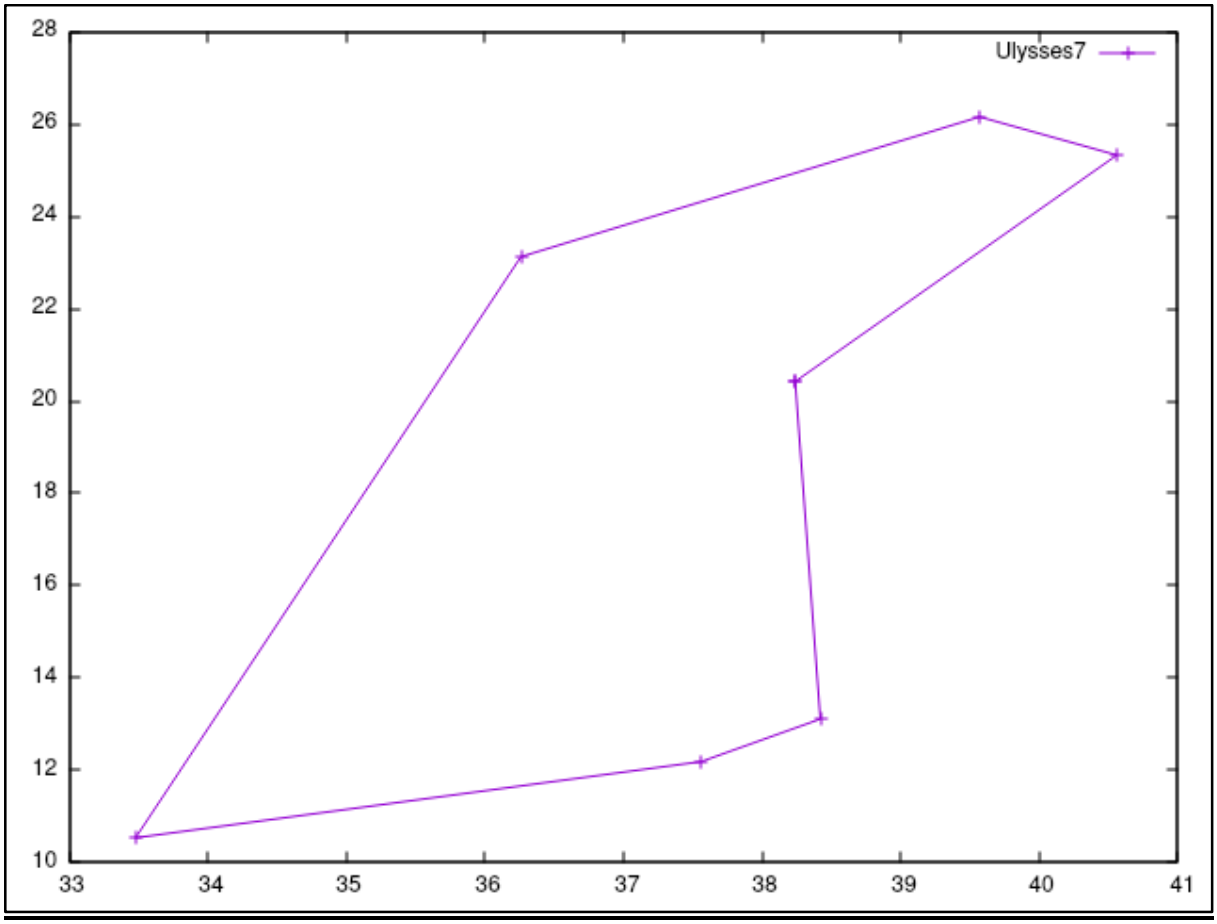


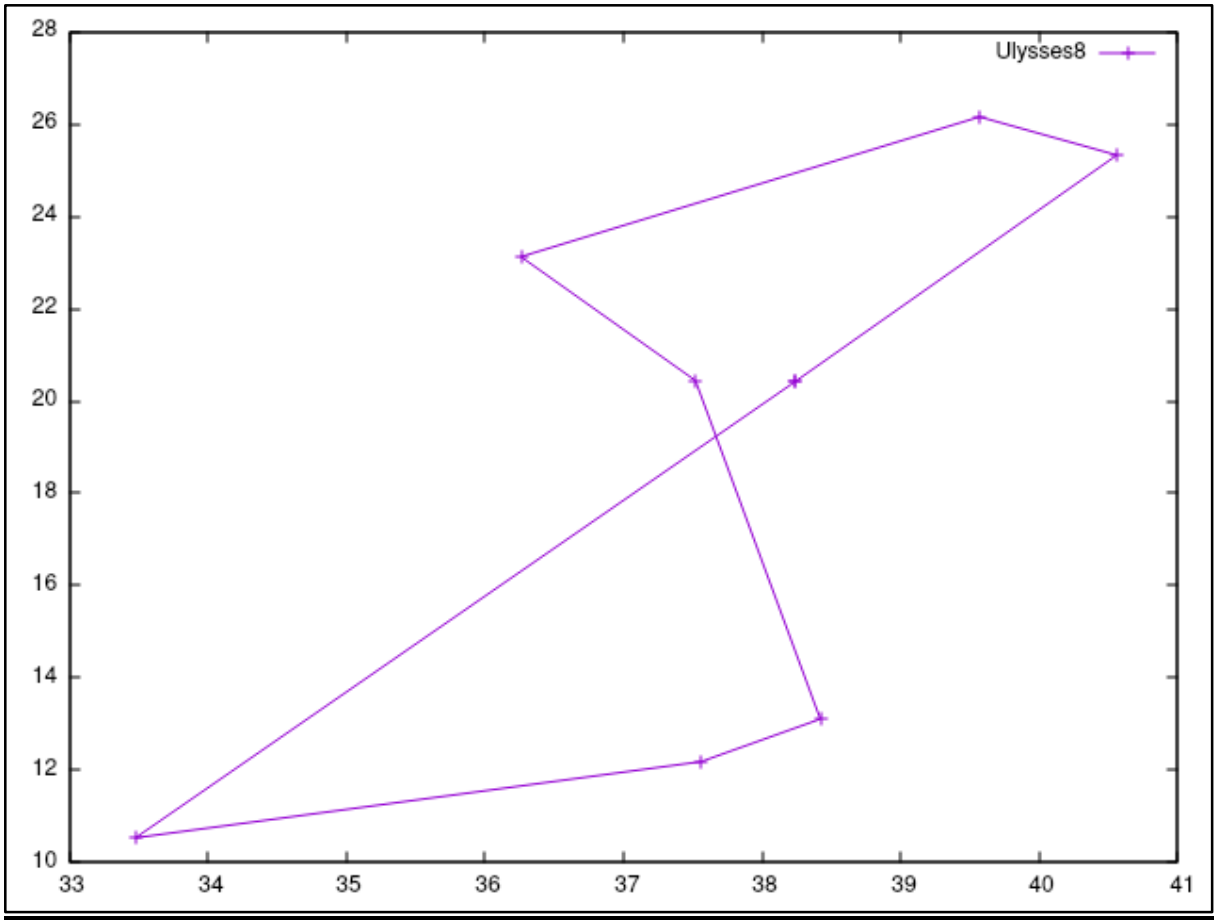
Las siguientes imágenes muestran el recorrido obtenido para el número de ciudades indicado anteriormente:

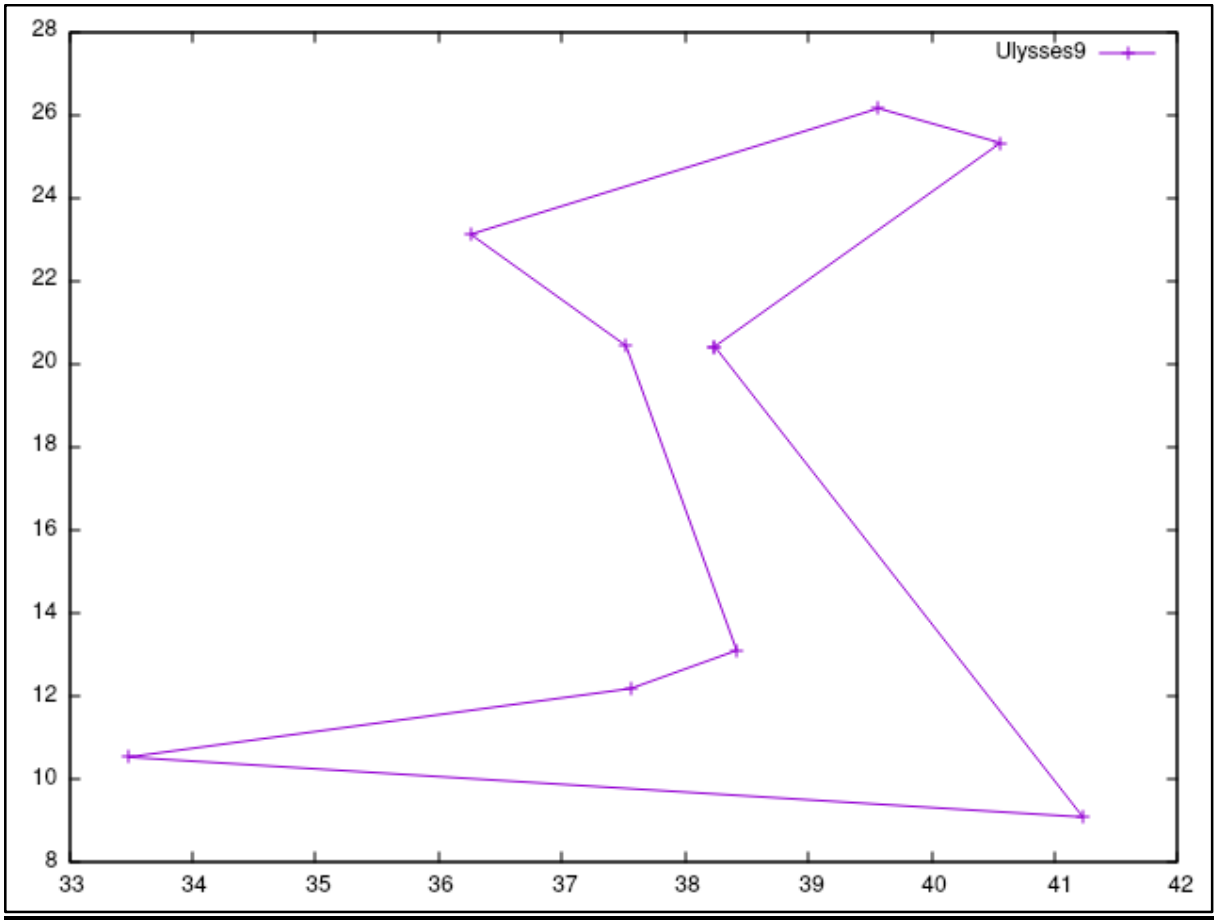


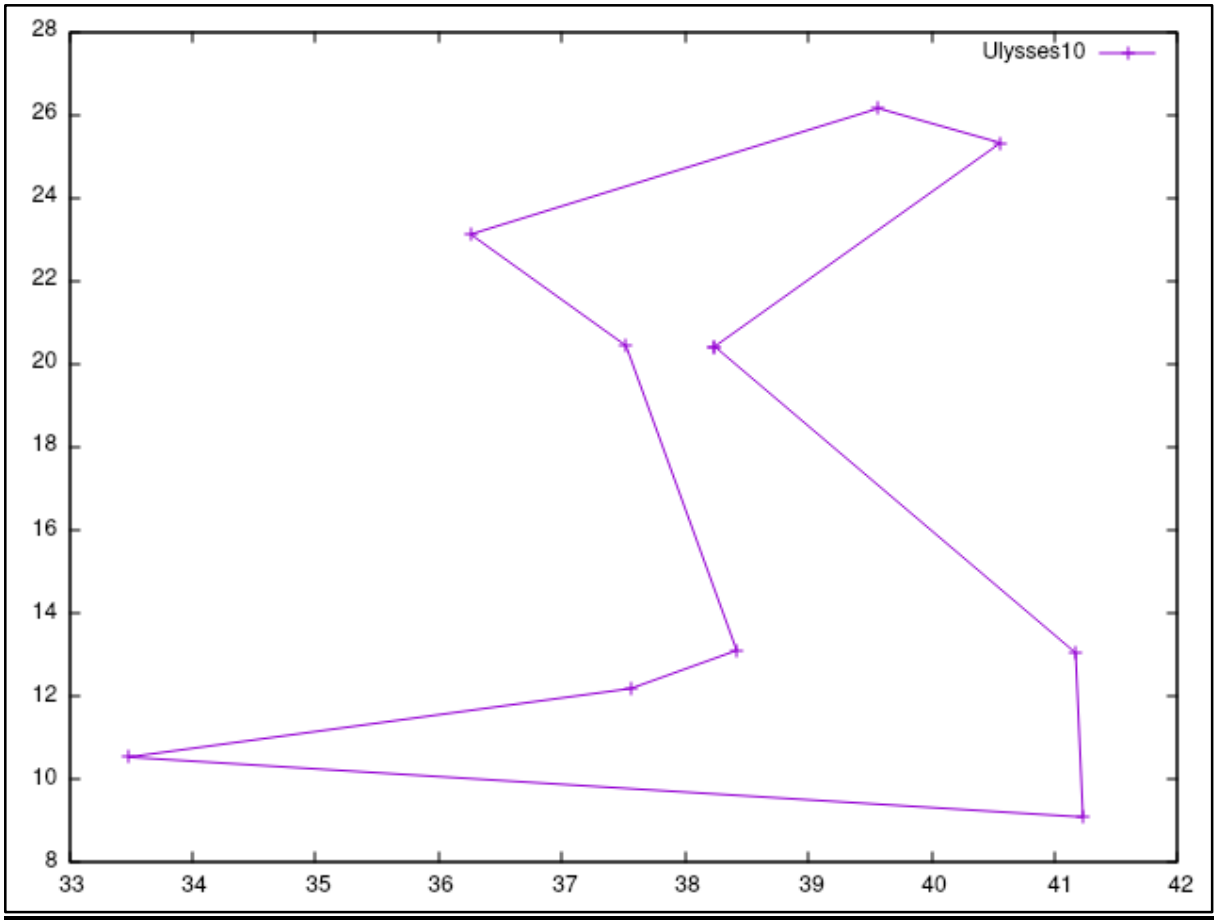


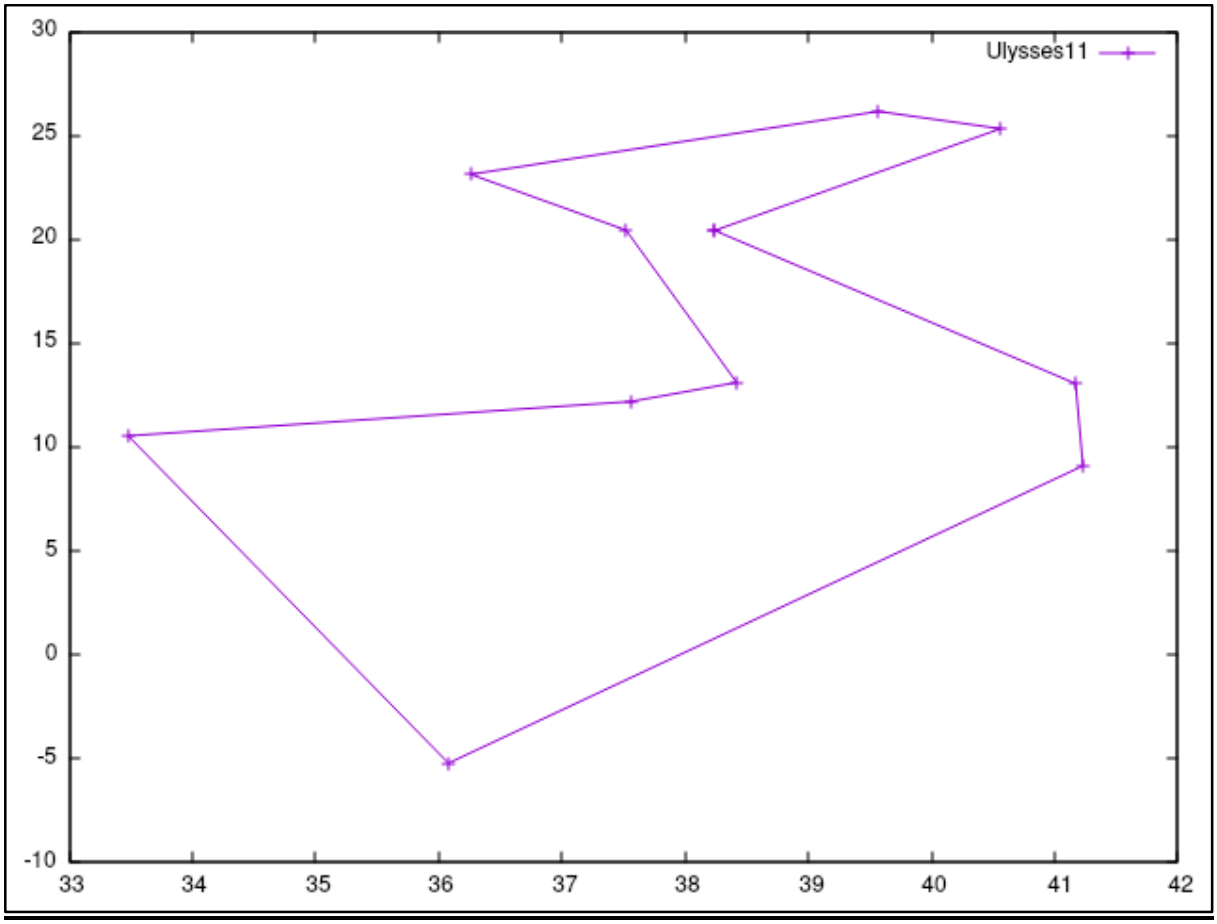

Ulysses6
28
26
24
22
20
18
16
14
12
10
33
34
35
36
37
38
39
40
41

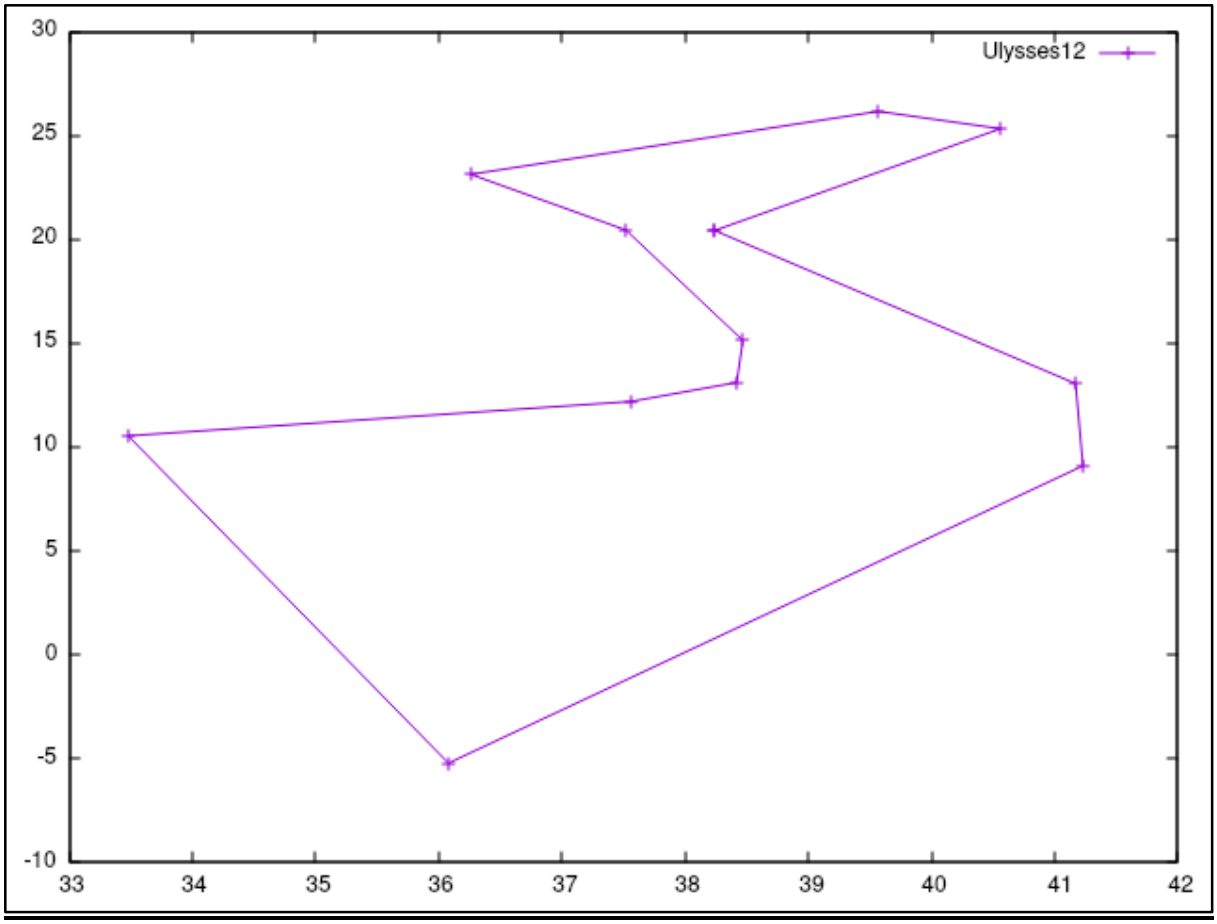

Ulysses7
28
26
24
22
20
18
16
14
12
10
33
34
35
36
37
38
39
40
41


Ulysses8
28
26
24
22
20
18
16
14
12
10
33
34
35
36
37
38
39
40
41


Ulysses9
28
26
24
22
20
18
16
14
12
10
8
33
34
35
36
37
38
39
40
41
42


Ulysses10
28
26
24
22
20
18
16
14
12
10
8
33
34
35
36
37
38
39
40
41
42


Ulysses11
30
25
20
15
10
5
0
-5
-10
33
34
35
36
37
38
39
40
41
42


Ulysses12
30
25
20
15
10
5
0
-5
-10
33
34
35
36
37
38
39
40
41
42

d. **Pseudocódigo**

PARÁMETROS INICIALES PARA LA LLAMADA B&B

- Nodo inicial (ciudad 0)
- Matriz de distancias.
- Nodo solución que almacena un vector con el recorrido solución.

LLAMADA A FUNCIÓN B&B

- Inicializamos los parámetros iniciales:
    - El nodo inicial con sus parámetros correspondientes.
    - Una Priority_queue llamada “cola” que almacena los nodos visitados inicializada con el nodo inicial.
    - Se inicializa cota global con la distancia obtenida mediante solución Greedy

- Mientras haya nodos en la Priority_queue:
    - Obtiene el nodo en el top de la cola con prioridad
    - Borra el tope de la cola con prioridad.

    - Para cada ciudad candidata a visitar
        - Crea un nodo hijo (ciudad candidata) a partir del padre (ciudad actual)
        - Actualiza el hijo con el nuevo coste hasta el padre
        - Si el coste no sobrepasa la cota global (poda si sobrepasa o iguala)
            - Actualiza el resto de parámetros del hijo
            - Si el nodo hijo es un nodo hoja:
                - Actualiza el hijo con el coste de volver a la ciudad origen
                - Si el coste final mejora la solución previa
                    - Actualiza la cota global
                    - Actualiza el recorrido solución

            - Si ese nodo **no** es un nodo hoja
                - Inserta el hijo en la cola con prioridad

6. **Conclusiones**

Para ambos problemas realizados, hemos visto que el tipo de problema es muy parecido, ya que ambos se basan en obtener una solución de ordenación, donde el algoritmo Backtracking para la cena de gala busca maximizar y el algoritmo Branch and Bound, minimizar la solución.

Hemos podido comprobar durante la realización de esta práctica, que ambos algoritmos, tanto Backtracking como Branch & Bound son bastante parecidos en cuanto eficiencia y resultados.

Sin embargo, ambos son menos eficientes que el algoritmo realizado por Programación Dinámica en la práctica anterior, como se ha podido comprobar en los tiempos de ejecución para Backtracking y Branch and Bound.

Mientras que con Backtracking llegamos a 14 comensales y con Branch & Bound a 12 ciudades, con la Programación Dinámica conseguimos llegar hasta 23 ciudades, ya que es mucho más eficiente y conseguimos reducir considerablemente el tiempo.

Mientras tanto, si comparamos con un algoritmo Greedy, podríamos ver como los algoritmos voraces nos permiten obtener una solución aproximada, que no óptima, en un tiempo muy reducido para tamaños de problema grandes, mientras que con Backtracking y Branch and Bound no es posible, ya que espacio de problema generado es muy grande y solo podemos limitarlos a tamaños de problemas pequeños, pero obteniendo su óptimo.

7. **Bibliografía**

Lecciones de Algorítmica - Verdegay Galdeano, José Luis
https://drive.google.com/drive/folders/1cHzJCVYdTdXyI7LFm0sGnqRb06Kz0mMJ