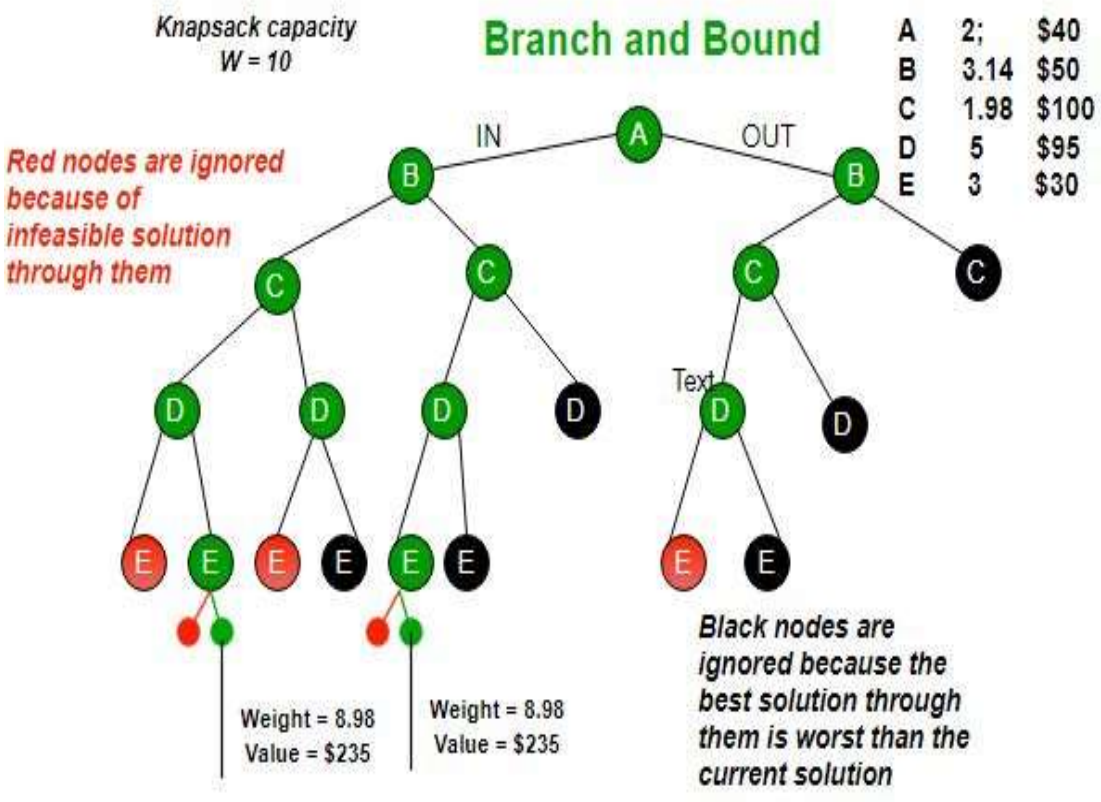
PRÁCTICA 5: BACKTRACKING Y BRANCH & BOUND

- Ahmed El Moukhtari Koubaa
- Damián Marín Fernández
- Eduardo Segura Richart
- Jesús Martín Zorrilla



ÍNDICE

1. Descripción del problema.

Para esta práctica se nos encarga la realización de 3 algoritmos. Debemos realizar un algoritmo basado en las técnicas de backtracing para la resolución del problema del viajante de comercio, por sus siglas en inglés TSP. Debemos, además, resolver el TSP, pero ahora usando ramificación y poda, en inglés branch and bound. El segundo problema y tercer algoritmo a realizar, es el problema de la ITV, mediante backtracking.

Una descripción detallada del tan conocido TSP sería: teniendo un conjunto de n ciudades debemos recorrer todas estas una sola vez y volviendo a la ciudad de origen, de manera que la suma de las distancias entre cada dos ciudades adyacentes sea mínima, es decir, debemos dar el recorrido de menor distancia para las n ciudades del problema.

La descripción del problema de la ITV es: teniendo m líneas de inspección de vehículos y n vehículos, debemos inspeccionar los n vehículos en el menor tiempo posible, teniendo en cuenta que cada vehículo tiene unas características particulares y por lo tanto un tiempo de inspección también particular.

Se nos pide dar para branch and bound resultados relativos como son: el número de nodos expandidos, el tamaño máximo de la cola con prioridad, número de veces que se poda y tiempo empleado para resolver el problema. Se nos sugiere además hacer comparaciones con resultados de otras prácticas.

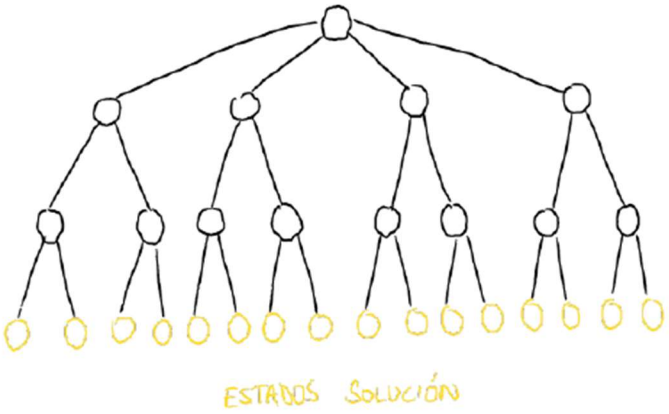
Finalmente, se nos pide una comparación y un estudio experimental de los programas realizados para la resolución del TSP, como son backtracking y branch and bound.

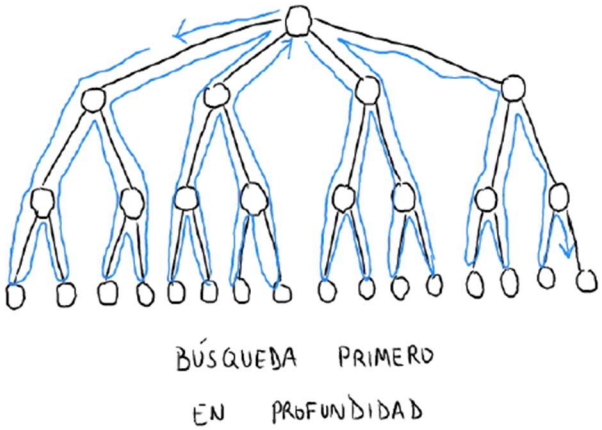
2. TSP usando Backtracking.

Descripción detallada del algoritmo.

El algoritmo "Backtracking" o "Vuelta atrás" es un método de búsqueda en grafos que realiza secuencias de decisiones hasta encontrar la solución correcta a un objetivo establecido. En este caso que nos ocupa, hemos desarrollado una versión del algoritmo Backtracking para encontrar una solución para el problema del Viajante de Comercio (TSP).

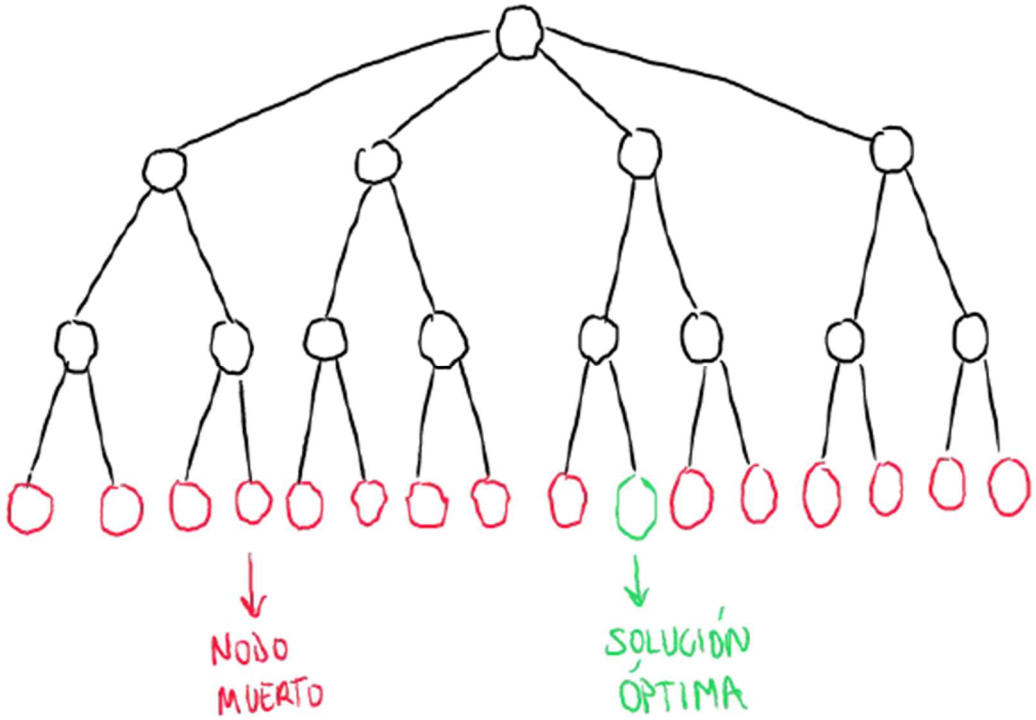
El algoritmo Backtracking realiza un recorrido del árbol primero en profundidad, esto quiere decir, permitiéndonos el símil, que recorre el árbol por "ramas" siendo los nodos "hoja" las posibles soluciones. Cuando se está explorando el árbol, se comprueba mediante una estimación si el camino que se está recorriendo lleva a una solución factible, es decir, si la solución que vamos a descubrir mejora la que ya se ha encontrado previamente. Si la estimación nos garantiza que el camino que estamos recorriendo en el árbol nos lleva hacia un nodo "hoja" (solución) factible, continuamos con la exploración. Sin embargo, si la estimación nos dice lo contrario, dicha "rama" que estamos explorando se poda y retrocedemos en el árbol en busca de otra "rama" que analizar.





Antes de comenzar a explicar nuestra implementación del algoritmo vamos a destacar varios aspectos importantes a tener en cuenta.
Supongamos que tenemos N = 5 ciudades:

- **Solución:** sea X una secuencia de ciudades (recorrido), una posible solución podría ser **X = [0,4,2,3,1,0]** junto con la distancia recorrida.
- **Restricciones explícitas:** X[i] pertenece a {0,…,N-1}
- **Restricciones implícitas:** Una ciudad no puede aparecer dos veces en el recorrido (salvo la ciudad de origen si esta aparece en primera y última posición del recorrido).
- **Espacio de Soluciones:** Permutaciones de N elementos.



Lo primero a destacar de la adaptación basada en Backtracking para el TSP es que nos encontramos ante un problema de minimización, es decir, nuestro algoritmo genera múltiples decisiones para encontrar aquella que logre el menor costo de recorrido total tras visitar todas las ciudades una única vez y regresar a la ciudad de origen. Por tanto, el algoritmo parte con una cota superior que se actualiza cada vez que se encuentra una solución de menor costo. Para ello, debemos considerar:

- Costo del camino que representa la solución parcial
- Un estimador (por debajo, debido a la minimización) del costo para ir al resto de ciudades.

Una vez destacados los aspectos a tener en cuenta, podemos comenzar con la explicación de nuestra implementación. En nuestra implementación existen tres elementos que son triviales en la solución del problema:

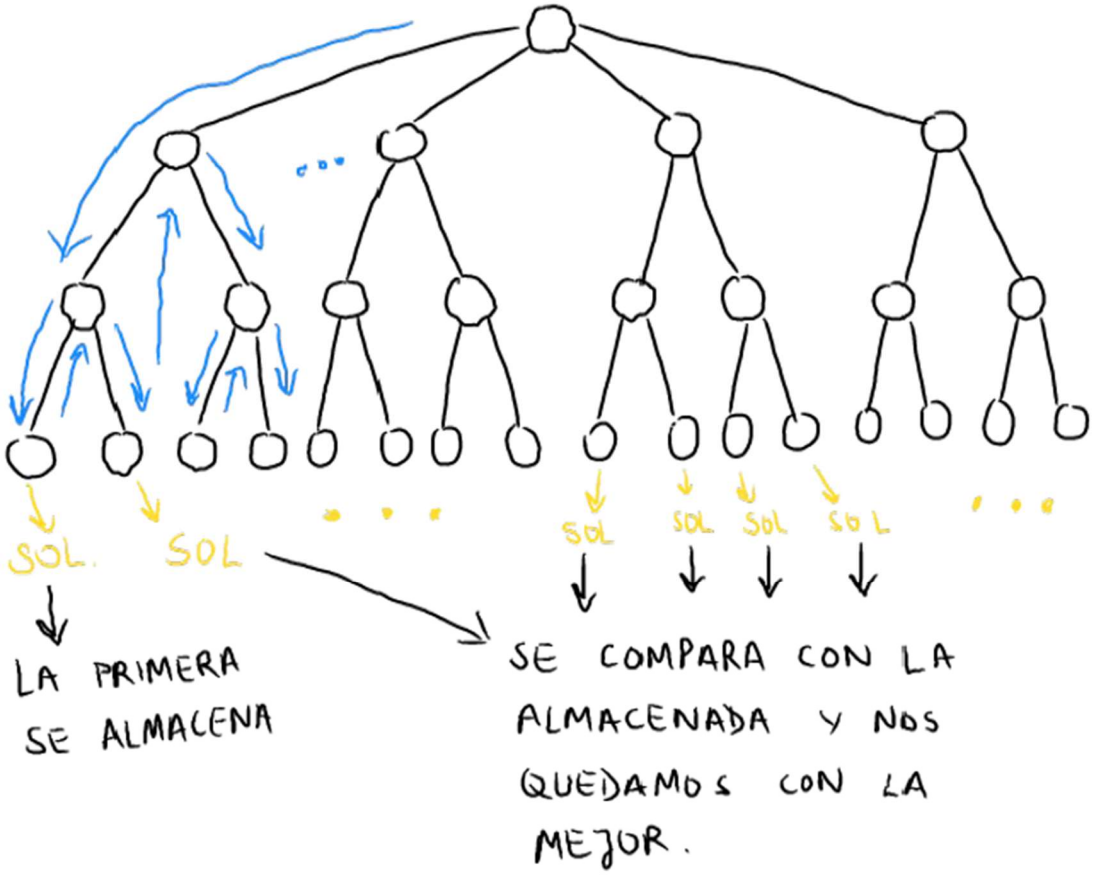
- **Dos vectores de índices de ciudades** que almacenan el recorrido parcial (de la solución parcial) y el recorrido definitivo (el de menor costo en el momento), respectivamente.
- **Un vector de booleanos** cuya dimensión es el número de ciudades y, por tanto, cada posición se corresponde con una ciudad. Dicho vector controla las ciudades que se han visitado. De esta manera, si el elemento de la posición 2 está a "true" quiere decir que ya se ha visitado la ciudad de índice 2, por ejemplo. Igualmente, si un elemento está a "false" quiere decir que la ciudad con dicho índice no se ha visitado.
- Una **función de factibilidad** que nos proporciona esa estimación que nos ayuda a saber si estamos analizando una "rama" que nos dirige a un nodo solución factible.

El esquema de ejecución del algoritmo es el siguiente:

1) Partimos del nodo origen (que se establece como **posición actual**), cada nodo representa una ciudad y sus hijos son las ciudades que puede visitar.

Excepción (primera ejecución) nodo ciudad origen:Este nodo se introduce en el vector de índices de ciudades y se establece a “true” la posición correspondiente al indice de la ciudad de origen en el vector de booleanos. El coste del nodo origen se ha establecido a 0 previamente.

2) Se comprueba si se ha llegado a un nodo “hoja” en cuyo caso se comprueba si la solución obtenida mejora la anterior. Si lo hace, se sustituye el coste y el recorrido. Si no, se descarta. Si el nodo actual es un nodo “hoja”, este punto termina la ejecución en el nodo actual.



3) Se consulta la función de factibilidad a partir de este nodo, es decir, si el camino que siguen los nodos hijos nos lleva a una solución factible. Si la función de factibilidad nos da “el visto bueno” se realiza lo siguiente **para cada nodo hijo** (ciudades que todavía no se han visitado visitar):

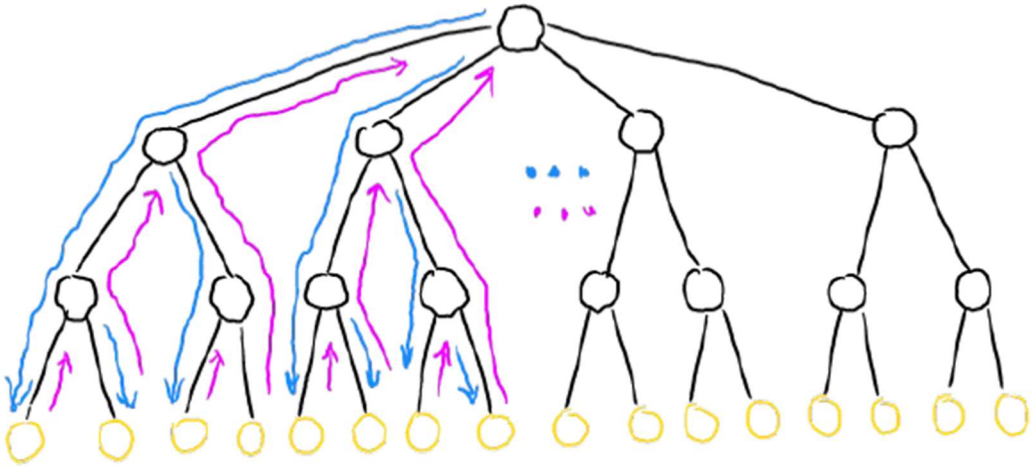
i. Se calcula la distancia que hay respecto al nodo actual y se acumula en el coste.
ii. Se añade su índice al vector del recorrido parcial.

iii. Se establece el elemento correspondiente a su índice a “true” en el vector de booleanos (ciudades visitadas)

iv. Se repite el proceso a partir del paso 1. Esto quiere decir que se baja un nivel en el árbol de búsqueda (**Recursividad**).

v. Se produce la **“vuelta atrás”**:

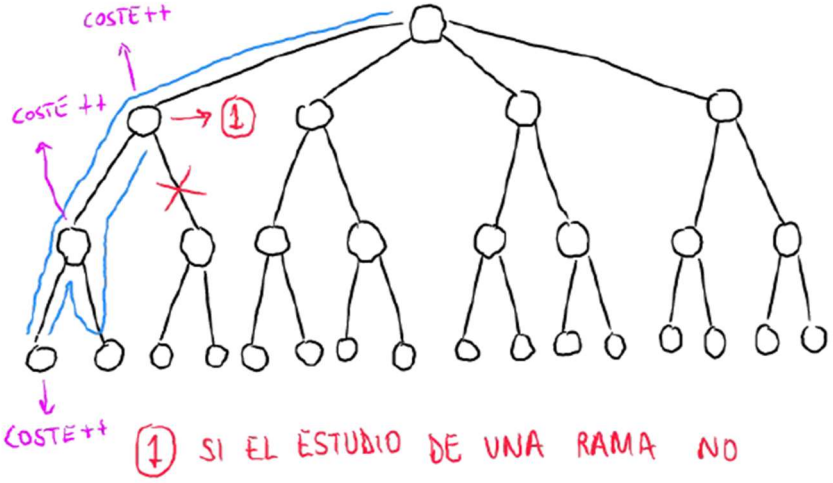
- Se elimina de visitados (false en el vector de booleanos).
- Se elimina del vector del recorrido.
- Se decrementa el coste añadido de su distancia con respecto al nodo que lo precede.



BACTRACKING

"VUELTA ATRÁS"

Si la función de factibilidad rechaza la exploración, se poda dicha “rama” (se termina la ejecución en este nodo).



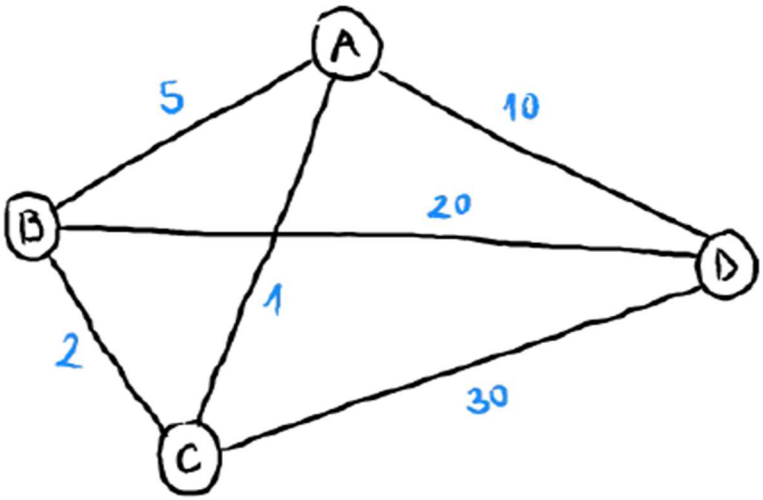
1 SI EL ESTUDIO DE UNA RAMA NO ES FACTIBLE, ESTA SE PODA

NO ES FACTIBLE CUANDO SE PREVÉ QUE NO APORTARÁ UNA SOLUCIÓN MEJOR.

Función de factibilidad:

La idea de la función de factibilidad es realizar el cálculo de un estimador, es decir, un valor numérico por el cual decidamos podar o proseguir con la exploración de la "rama" en la que nos encontramos. En nuestro algoritmo de Backtracking, la función de factibilidad se ha diseñado basándose en la cota del mínimo costo de "Salir". ¿Qué quiere decir esto? El mínimo costo de "Salir" es el mínimo valor de todas las entradas no nulas de la matriz de costos (o matriz de distancias). En otras palabras, es el valor del arco de menor costo que sale de un nodo. De esta manera, para calcular el estimador, lo que haremos es que para cada nodo no visitado restante, obtenemos el costo del arco que menor valor tenga saliendo de este (mínimo costo de "Salir") y lo acumulamos. Este cálculo también lo haremos con el nodo actual (padre de los nodos restantes). Si el valor del estimador es menor que el costo de la solución actual, continuamos con la exploración. Si no, se poda la "rama".

En la siguiente imagen podemos ver un ejemplo de cálculo de estimador.



A - B → 5

SALIR B → 2

SALIR C → 1

SALIR D → 10

ESTIMADOR = 5 + 2 + 1 + 10 = 18

En la imagen se puede apreciar un grafo cuyos nodos representan ciudades. En este ejemplo, el algoritmo ya ha explorado los nodos A y B y faltan por explorar los nodos C y D. A continuación, se procede a realizar el cálculo del estimador de la función de factibilidad.

En color verde podemos observar que se calcula el mínimo costo de "Salir" del nodo actual. En este caso, dicho valor es 5.

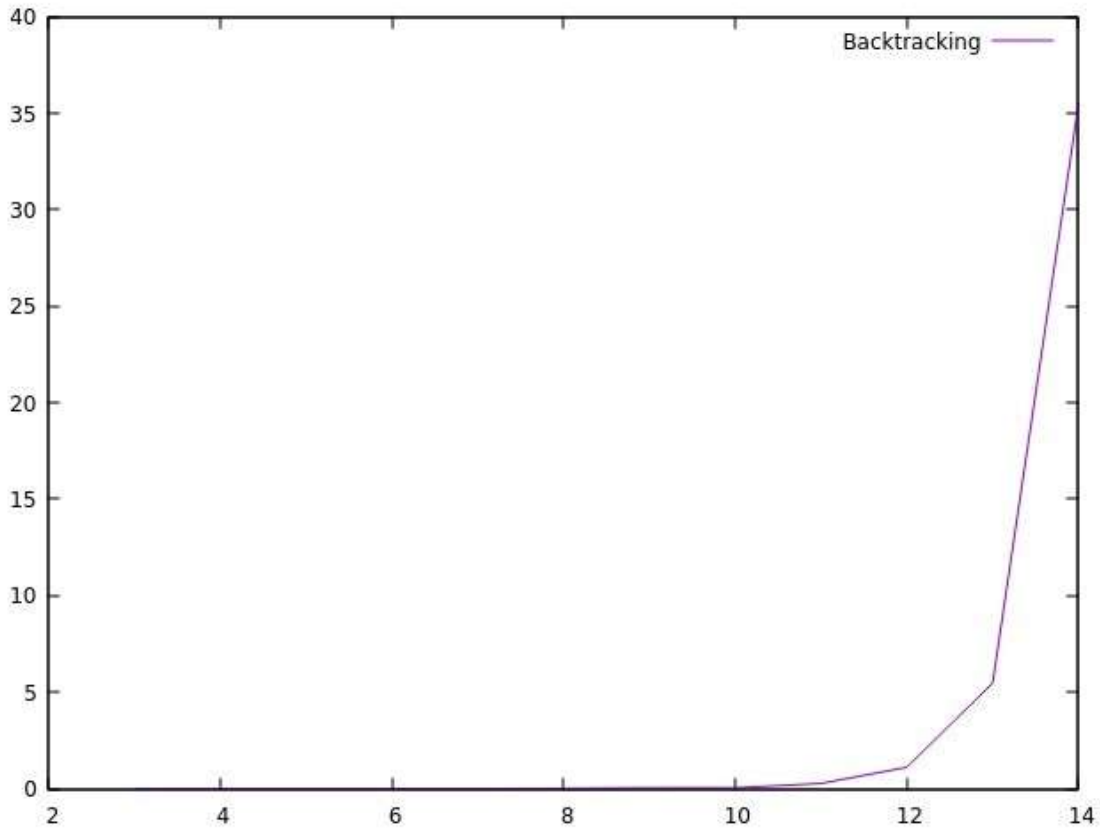
En rojo, lo mismo correspondiente a cada nodo hijo. Para el nodo B es 2, para el nodo C es 1 y para el nodo D es 10.

Todo ello (verde y rojo) hace la suma de 18 que sería el valor del estimador y que se compararía con la solución actual.

Cálculo de la eficiencia empírica

Para el cálculo de la eficiencia empírica se ha ejecutado el algoritmo para los siguientes números de ciudades:

Número de ciudades	Tiempo(Seg)
3	8.358e-06
4	2.0328e-05
5	8.1754e-05
6	0.000712842
7	0.00212153
8	0.00620664
9	0.0408645
10	0.0407463
11	0.266186
12	1.09977
13	5.46333
14	35.473



Caso de ejecución

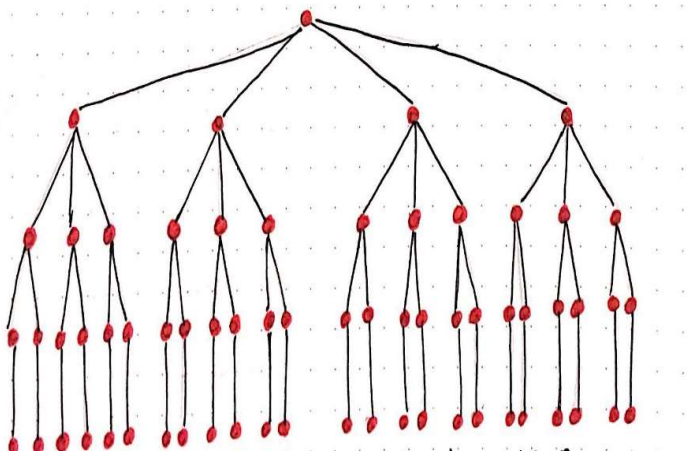
Para el caso de ejecución, se ha realizado la prueba con las 14 primeras ciudades del fichero “ulysses16.tsp” para el que se han conseguido los siguientes resultados:

```
edu@edu-Lenovo-ideapad-510-15ISK:~/Facultad/2º/alg/Backtracking$ g++ -std=gnu++0
x tsp.cpp -o tsp
edu@edu-Lenovo-ideapad-510-15ISK:~/Facultad/2º/alg/Backtracking$ ./tsp ulysses14
.tsp
Numero ciudades: 14
El recorrido (15) tomado es:
0 38.24 20.42
13 37.51 15.17
12 38.15 15.35
11 38.47 15.13
6 38.42 13.11
5 37.56 12.19
4 33.48 10.54
10 36.08 -5.21
8 41.23 9.1
9 41.17 13.05
2 40.56 25.32
1 39.57 26.15
3 36.26 23.12
7 37.52 20.44
0 38.24 20.42

La distancia recorrida es: 70.903
edu@edu-Lenovo-ideapad-510-15ISK:~/Facultad/2º/alg/Backtracking$
```

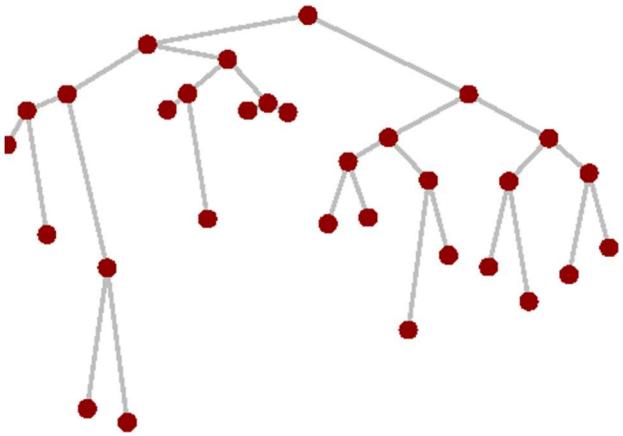
3.TSP usando branch and bound.

Descripción detallada del algoritmo.

Nuestro algoritmo branch and bound es “sencillo” pero bastante potente. Lo que se busca es resolver de manera viable un problema NP-duro como es el problema del viajante de comercio. La forma de hacer esto obteniendo un resultado óptimo es reducir el número de cómputos que se realizan para que sea viable. En branch and bound esto se consigue siguiendo una estrategia de ramificación que nos permitirá concentrarnos en las “ramas” más prometedoras y la poda de posibles resultados que se estiman como poco prometedores.



Antes de empezar la descripción detallada de nuestro algoritmo, entendamos mejor la metodología branch and bound. La potencia de esta técnica reside en varias claves como son la estimación y las cotas superiores e inferiores (en nuestro caso solo usaremos una cota inferior, ya que el TSP es un problema de minimización). Así pues, branch and bound es un método general de búsqueda que explora un árbol comenzando desde la raíz y aplicando entonces procedimientos de cotas inferiores y superiores que, si se cumplen concluyen en la finalización del algoritmo, pero en otro caso dan lugar a una ramificación y dividiendo así el problema en dos o más subproblemas, este método se aplica recursivamente a los subproblemas.



Es importante recalcar que cuando se encuentra una solución optimal para un subproblema esta será factible para el problema completo, pero no necesariamente el óptimo global. A partir de la primera solución encontrada se actualizan los valores de las cotas y para las estrategias de ramificación y poda se tendrán en cuenta estas. Una vez se han examinado o podado todos los nodos, se termina la búsqueda y se devuelve el mejor resultado encontrado, el cual es optimal.

Una vez aclarado lo anterior ya podemos comenzar la descripción detallada de nuestro algoritmo:

1) Partimos del nodo origen, cada nodo tiene una ciudad y un conjunto de ciudades por recorrer, y lo añadimos a la cola con prioridad.
2) Mientras la cola no esté vacía, tomamos el primer nodo de esta, que es el más prometedor.
3) Para toda ciudad del conjunto estimamos su coste, para ello:
    a. Para toda ciudad sin recorrer (excluyendo la ciudad que se está estimando e incluyendo la ciudad de origen) se busca aquella ciudad del conjunto cuya distancia sea mínima, es decir, sea la más cercana.
    b. Una vez halladas todas esas distancias mínimas, se suman.
    c. A la suma de las distancias mínimas se suma el costo acumulado de llegar al nodo que se está estimando, es decir, la suma de la distancia entre el nodo anterior y el estimado (como este proceso se repite al final el acumulado consiste en la suma de las distancias del camino que desemboca en el nodo estimado).
4) Tras estimar coste, introducimos aquellos nodos cuya estimación sea menor o igual que el coste mínimo a una cola con prioridad ordenada de menor a mayor según la estimación.
5) Si no quedan ciudades por recorrer hemos encontrado entonces una solución y si el coste de este camino es menor que el coste mínimo, se actualizan el coste mínimo y el recorrido mínimo.

En resumen, tomamos el nodo origen y lo introducimos a una cola, mientras la cola no esté vacía vamos tomando el elemento más prometedor y estimando el coste de sus hijos, cuando no quedan ciudades por recorrer actualizamos los mínimos si corresponde y repetimos nuevamente.

Aunque no aparezca reflejado en el pseudocódigo, sí estamos calculando información relativa como el tamaño máximo de la cola, los expandidos y los podados.

El coste mínimo se puede inicializar a infinito o a valores como el resultado de aplicar Greedy, dependiendo de esto los tiempos de ejecución cambiarán pues se podrá comenzar a podar antes y ramificarse menos.

La explicación que acabamos de dar es un poco abstracta y muy conceptual, no llega a aclarar el algoritmo tanto como nos gustaría, así que vamos a dar una representación gráfica del problema y resolver para las 4 primeras ciudades de ulysses16 el TSP.

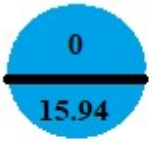
La matriz de distancias es la siguiente:

0 5.88233 5.42148 3.34819

5.88233 0 1.2919 4.48743

5.42148 1.2919 0 4.83011

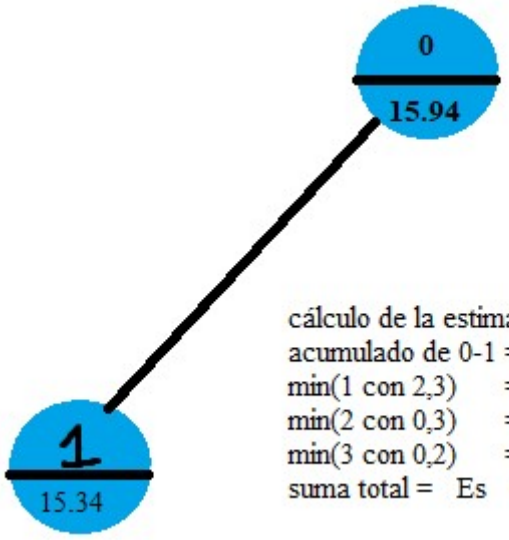
3.34819 4.48743 4.83011 0



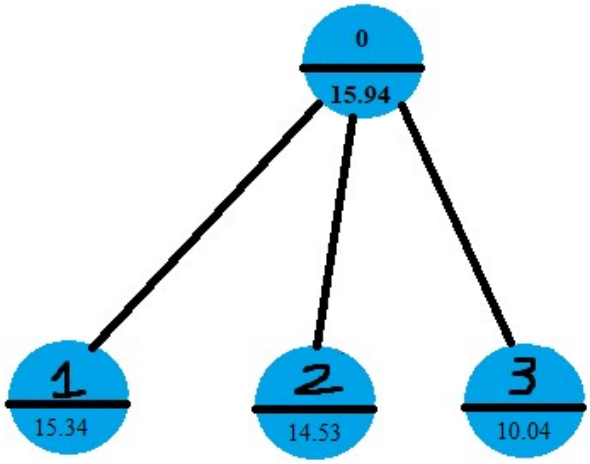
cálculo de la estimación
acumulado = 0
min(0 con 1, 2, 3) = 3.34
min(1 con 0, 2, 3) = 1.29
min(2 con 0, 1, 3) = 1.29
min(3 con 0, 1, 2) = 3.34
suma total = Es = 15.94

Calculamos el coste estimado para el nodo origen. Este coste consiste en la suma de las distancias mínimas de cada ciudad del conjunto.

Vamos ahora a hacer la estimación para los nodos aún no recorridos. En la imagen de la derecha vemos la estimación para el nodo de la ciudad 1.

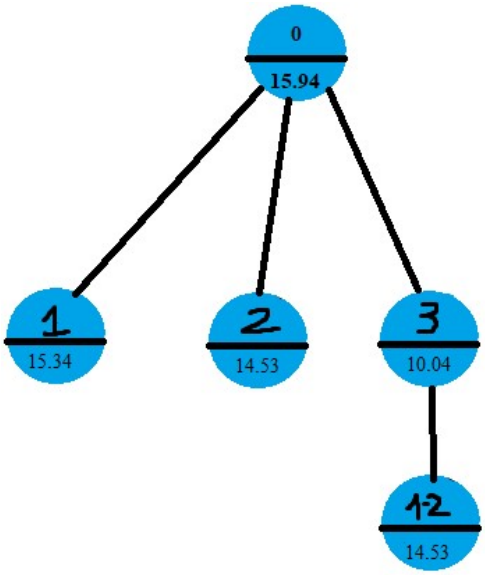


cálculo de la estimación
acumulado de 0-1 = 5.88
min(1 con 2,3) = 1.29
min(2 con 0,3) = 4.83
min(3 con 0,2) = 3.34
suma total = Es = 15.34

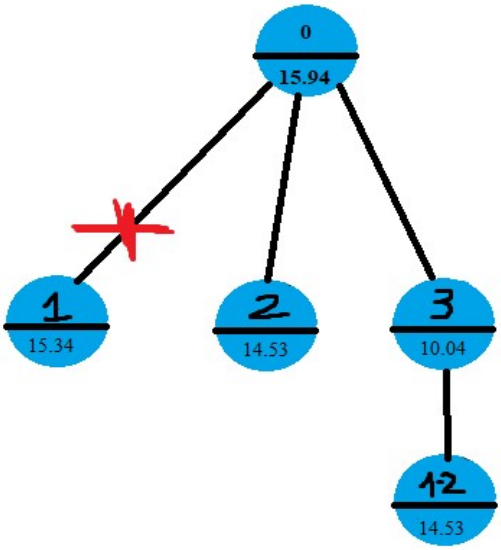


Tras expandir todos los nodos del primer nodo y calcular su estimación, los meteríamos a una cola con prioridad, donde el orden sería: {3,2,1}. Veamos el siguiente paso.

Al expandir el siguiente nodo en la lista nos quedaría solo el nodo “2” en el conjunto de no recorridos, por eso lo recorremos en esta expansión. Hemos encontrado un resultado y por eso podemos podar los nodos cuyas estimaciones sean mayores que el coste mínimo (ahora 14.53 antes infinito). El resultado sería el siguiente:

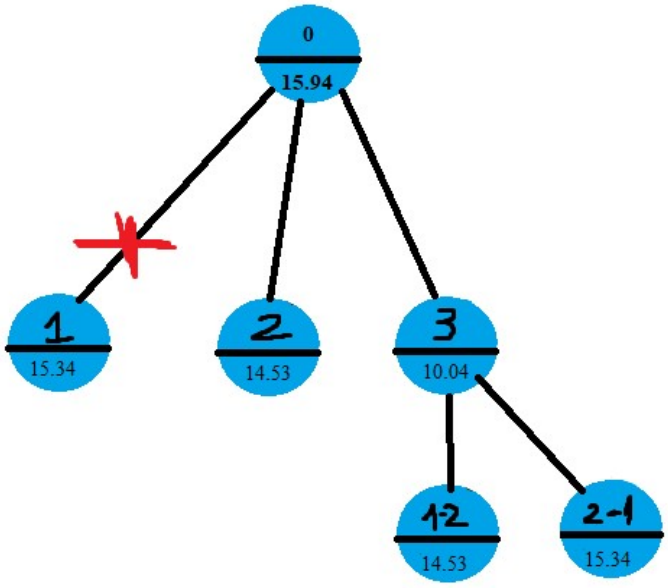


Cálculo del coste estimado acumulado 0-3-1-2-0 = 7.82+1.29+5.42 = 14.53
min(1 con 2) = 1.29
min(2 con 0) = 5.42
suma total = Es = 14.53



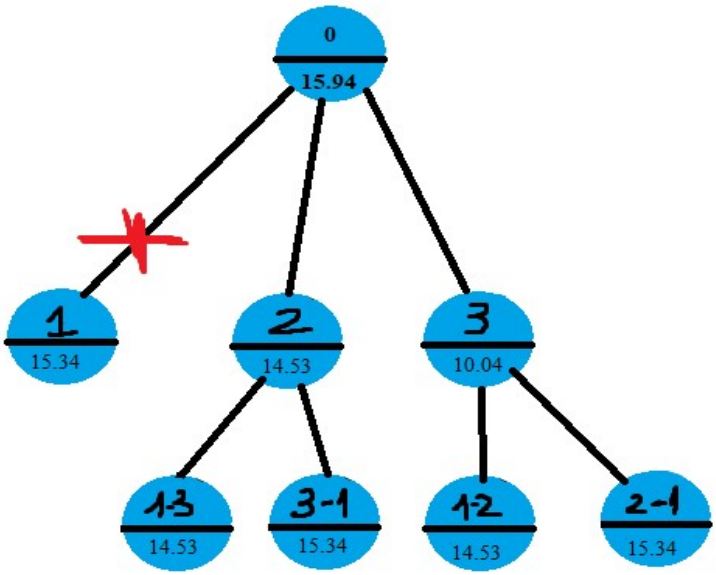
Cálculo del coste estimado acumulado 0-3-1-2-0 = 7.82+1.29+5.42 = 14.53
min(1 con 2) = 1.29
min(2 con 0) = 5.42
suma total = Es = 14.53

Hemos podado el nodo 1 que desciende de 0, es decir, el nodo 0-1. La cola quedaría ahora así {3, 2}. Debemos seguir ramificando 3 y luego comenzar con 2.



Tenemos ahora otro camino, pero este no mejora de ninguna forma al mejor anterior. El coste mínimo sigue siendo 14.53, inspeccionamos el siguiente elemento de la cola

Tras explorar el nodo 2, encontramos un camino que iguala al mejor anterior. Este camino es 0-2-1-3-0, iguala a 0-3-1-2-0, que es el anterior. Si nos fijamos en la matriz de distancias vemos que es simétrica, así que da igual ir en un sentido que en otro. Como el último camino no tiene un coste menor que la cota mínima, entonces nos quedamos con el anterior.

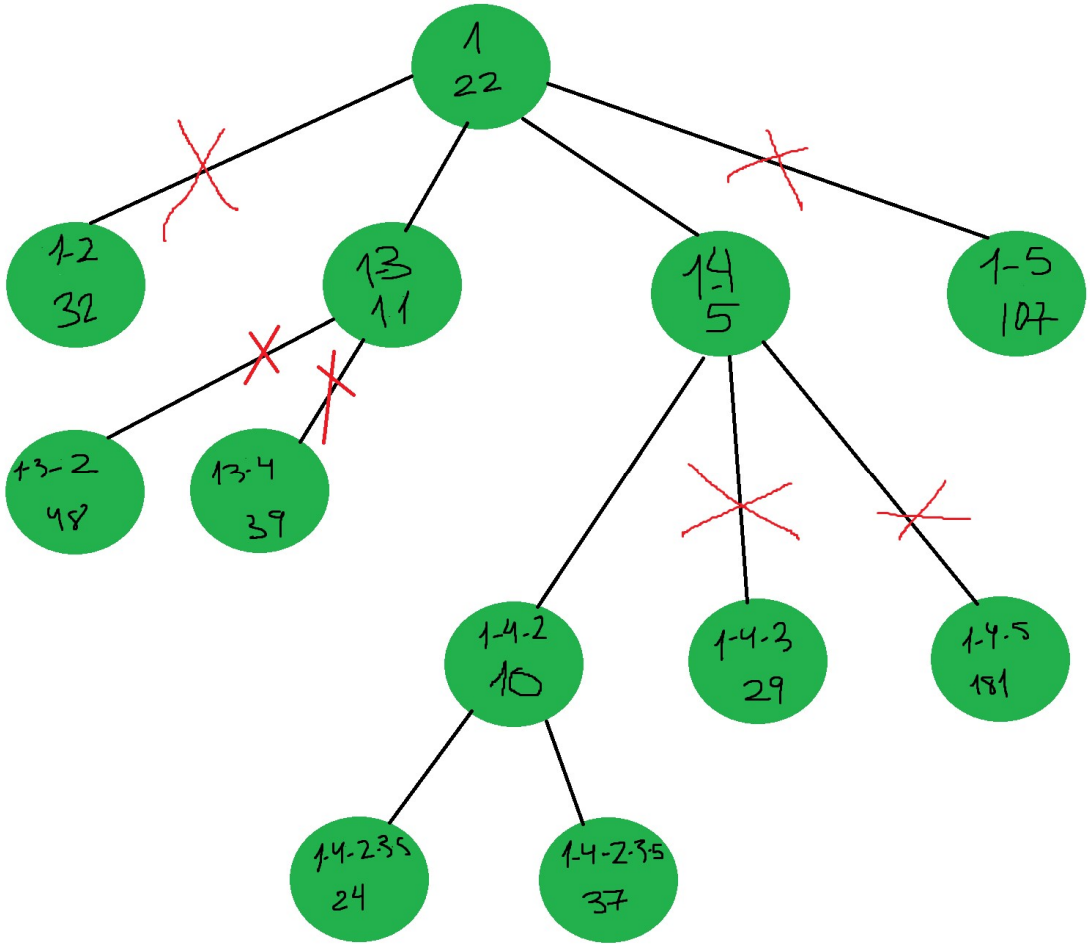


Veamos este caso de ejecución en nuestra terminal, con el programa implementado.

```
ahmed@OMEN:/mnt/d/ING INFÓRM
Distancias entre ciudades
0 5.88233 5.42148 3.34819
5.88233 0 1.2919 4.48743
5.42148 1.2919 0 4.83011
3.34819 4.48743 4.83011 0


4 8.26e-05
El coste minimo es: 14.549
0 3 1 2 0
ahmed@OMEN:/mnt/d/ING INFÓRM
```

Obtenemos la misma matriz, con los mismos resultados (a excepción de los redondeos). Así pues, hemos demostrado el correcto funcionamiento de nuestro algoritmo.

A continuación, se muestra una imagen en la que se intenta demostrar la principal diferencia con backtracking y es que branch and bound no hace búsqueda en profundidad sino más bien una búsqueda en anchura, en la que la cola se introducen nodos de distintos niveles y estos se recorren por orden de coste estimado, por lo que no importa el nivel al que pertenezca un nivel ni si se ha llegado más o menos lejos en la expansión de un nodo.

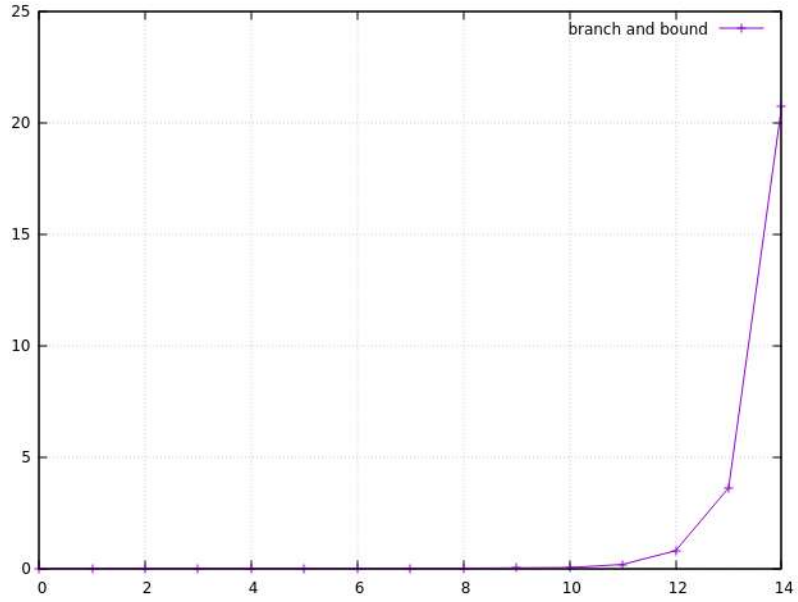


En este caso la solución es 1-4-2-3-5. Esta solución se encuentra después de haber expandido nodos de otros niveles antes y se sigue buscando en otros niveles tras ello.

Cálculo de la eficiencia empírica

Para llevar a cabo este estudio experimental, vamos a ejecutar nuestro programa implementado varias veces para tamaños entre 3 y 14.

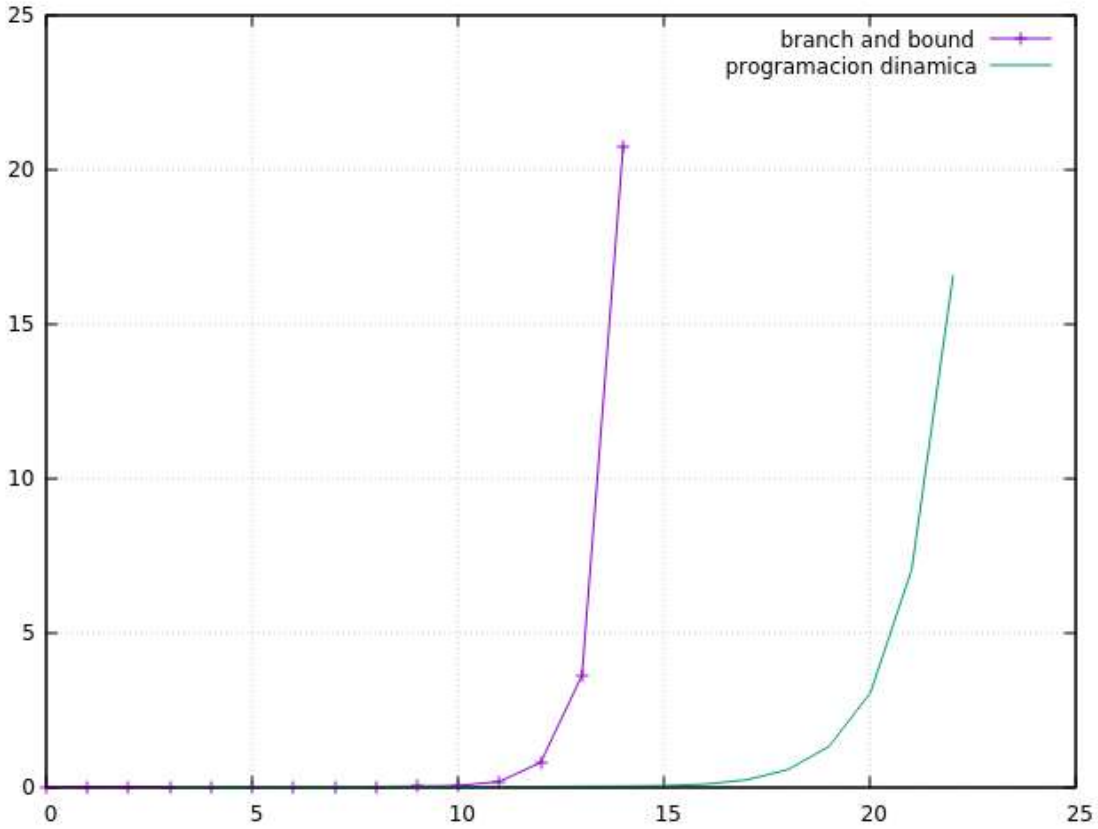
Número de ciudades	Tiempo (seg)	Tamaño máximo de la cola	expandidos	podados
3	4.18e-05	2	5	0
4	7.1e-05	6	11	4
5	0.0002671	24	35	23
6	0.0009614	100	100	99
7	0.0038856	325	233	324
8	0.0105989	1051	631	1050
9	0.0552644	5321	2802	5320
10	0.180636	17577	7033	17576
11	0.809554	79093	25452	79092
12	3.5446	330621	94452	330620
13	16.3346	1466816	377942	1466815
14	86.1361	7291518	1730204	7291517



Es evidente que la eficiencia no es factorial, esto porque podamos muchas ramas, aunque eso dependerá de la entrada del programa, sin embargo, sigue siendo una eficiencia exponencial.

Comparación con PD

Es evidente que branch and bound es mucho más ineficiente que programación dinámica, al menos para la entrada que hemos decidido usar (ulysses16), esto no tiene por qué ser así siempre y en todos los casos. Branch and bound consume menos memoria que programación dinámica.



Caso de ejecución

El caso mínimo para el que podemos ejecutar este programa es ulysses16, pero para más de 14 elementos el programa nunca llega acabar, por eso lo comprobaremos para las 14 primeras ciudades de ulysses16.

```
14 20.0796 7291518 1730204 7291517
0 13 12 11 6 5 4 10 8 9 2 1 3 7 0
1 14 13 12 7 6 5 11 9 10 3 2 4 8 1
```

En la primera secuencia los índices comienzan a ser enumerados en 0, en la segunda desde 1, es el mismo resultado, pero con diferente enumeración.

4.Problema de la ITV.

Descripción detallada del algoritmo

Para resolver este algoritmo debemos de aplicar una técnica de backtracking. Para ellos, en primer lugar y utilizando un enfoque Greedy calculamos una solución óptima, para tener una primera cota, usándola de referencia podemos centrarnos en analizar solamente las soluciones que sean más óptimas que nuestro primer enfoque Greedy.

El enfoque Greedy que vamos a usar consiste en lo siguiente: primero vamos a recorrer en orden los vehículos, y se mandaría a la cola el vehículo cuyo tiempo sea menor. Esto se hace para conseguir una solución de forma más rápida, desechando los resultados inútiles.

El código resultante es el siguiente:

```
20
21 void back_recursivo(solucion &s) {
22     if(s.size()<s.num_vehiculos()) {
23         int lin=0;                          //contador para iterar sobre las líneas
24         s.añadir_vehiculo();                //añadimos un vehiculo a la linea 0
25         while(lin<s.num_lineas()) {
26             if(s.correcto()) {
27                 back_recursivo(s);
28                 s.borrar_vehiculo(); //elimina el vehiculo temporal creado en back_recursivo
29             }
30             s.ulitmo_vehiculo_cambio(); //esto es para seguir explorando soluciones cambiando la
   línea
31             lin++;      //cambiamos de linea
32         }
33     }
34     else {      //tenemos que comprobar que la solucion sea correcta
35         s.solucion_actualizada();   //se actualiza la solucion y la cota
36         s.añadir_vehiculo(); //Añade un vehiculo extra
37     }
38 }
39
40
```

Lo que hemos hecho es crear una clase llamada solución, que contiene 3 vectores de ente-ros, con un tamaño igual al del número de vehículos. El primero de ellos nos indica la solución, otro lo usamos para ir calculando soluciones parciales y el último de ellos que usaremos para almacenar el tiempo que ha tardado cada vehículo. Esta clase también va a almacenar el número de líneas totales, y una cota obtenida inicialmente mediante un enfoque Greedy.

Cálculo de la eficiencia teórica

Teniendo en cuenta el enfoque con el que hemos resuelto el problema, notamos que siempre depende de la solución inicial tomada, por lo que no podemos conocer a ciencia cierta la eficiencia teórica de este algoritmo. En el mejor de los casos la solución óptima nos la dará el enfoque Greedy, mientras que el peor de los casos vendrá por una versión del algoritmo de fuerza bruta. Su eficiencia se encuentra entonces entre Ω (n*m) en el mejor de los casos y O(m^n), con n igual al número de vehículo y m igual al número de líneas.

Cálculo de la eficiencia empírica

Para tomar los datos se ha usado el siguiente script:

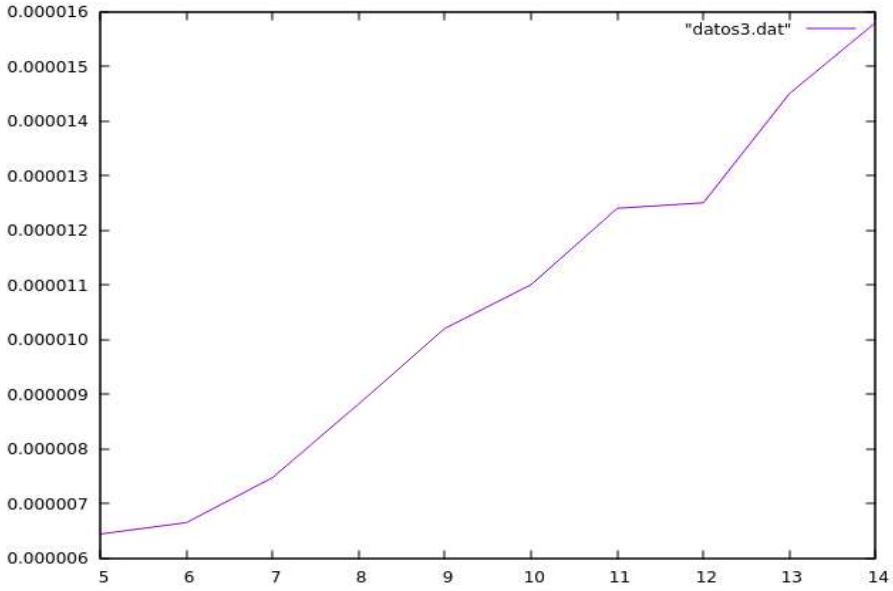
Se ha tomado el tiempo que tarda el algoritmo para 3 números de líneas distintos, 3, 5 y 7 y también se han tomado los datos utilizando una optimización O3. El número de coches que se ha usado va desde 5 hasta 14.

```
#!/bin/bash


for ((i=5; i<=14; i++))
do
    ./main $i 3
done

for ((i=5; i<=14; i++))
do
    ./main $i 5
done

for ((i=5; i<=14; i++))
do
    ./main $i 7
done
```

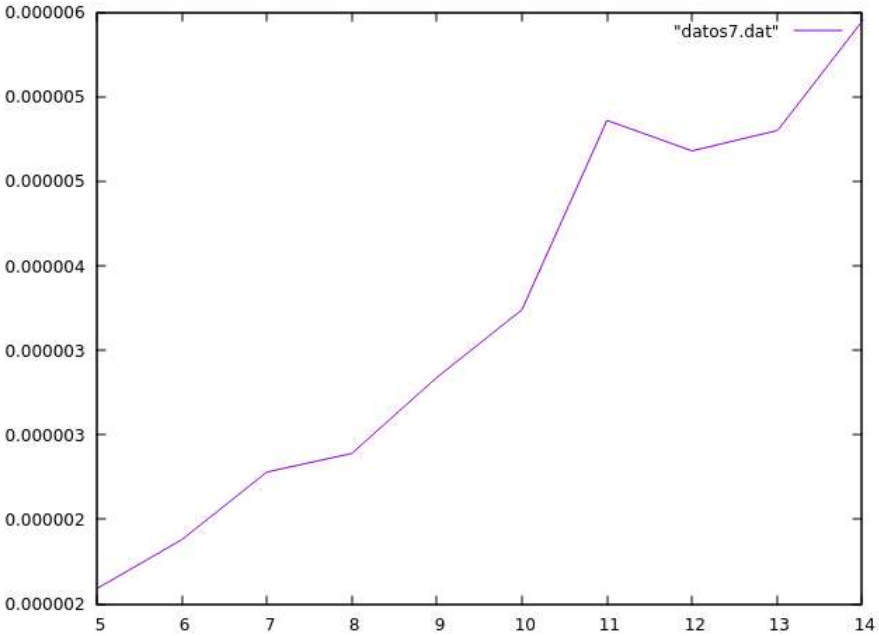
Las tablas resultantes son las siguientes:

Líneas	**3**	**5**
Vehículos	Tiempo(seg)	Tiempo(seg)
5	6.438e-06	2.09E-06
6	6.647e-06	2.38E-06
7	7.47E-06	2.78E-06
8	8.82E-06	2.89E-06
9	1.02E-05	3.34E-06
10	1.10E-05	3.74E-06
11	1.24E-05	4.86E-06
12	1.25E-05	4.68E-06
13	1.45E-05	4.80E-06
14	1.58E-05	5.45E-06

La gráfica para las tablas que usan 3 líneas:



La gráfica para las tablas que usan 7 líneas:



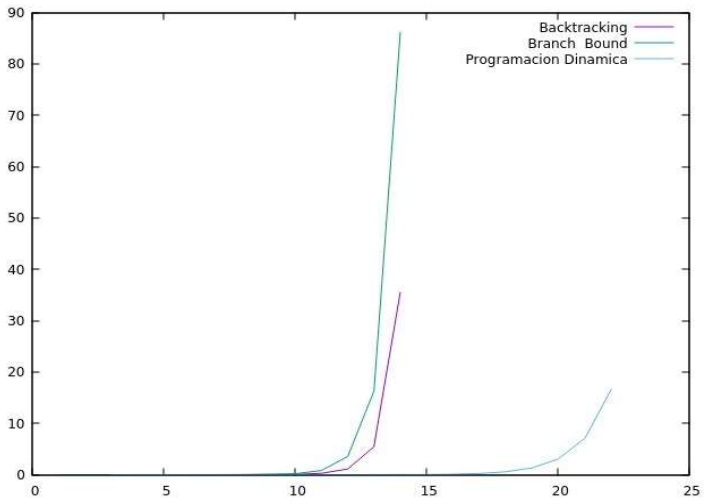
Cálculo de la eficiencia híbrida

Como ya hemos explicado en la eficiencia teórica, no podemos determinarla claramente, por lo que esta eficiencia no es calculable de una forma totalmente correcta, por lo que no se ha realizado el análisis correspondiente.

5. Conclusiones

Tras la realización de esta práctica sacamos como conclusión que tanto los métodos de backtracking como los métodos de branch and bound nos permiten resolver problemas de complejidad exponencial en un tiempo inferior al que obtendríamos dando una solución por fuerza bruta, para TSP e ITV.

Viendo las versiones branch and bound y backtracking para la resolución del TSP, nos damos cuenta que ambas tienen una complejidad exponencial. Los tiempos de ejecución de ambos algoritmos mejoran sus respectivas eficiencias teóricas. En ambos algoritmos la entrada, es decir, la secuencia de ciudades a recorrer, va a determinar el tiempo de ejecución sin depender necesariamente del tamaño, por ejemplo, para 50 ciudades podría llegar a tardarse menos que para 20 ciudades siendo estas diferentes. Ambos algoritmos van a depender mucho de las cotas y las estrategias usadas para la exploración, además de los criterios para llevar a cabo las estimaciones, por ejemplo, en branch and bound si inicializamos el coste mínimo empezaremos a podar ramas cuando encontramos al menos una solución, mientras que, si lo inicializamos al coste del recorrido mínimo Greedy, empezaremos a podar desde el primer momento.



Para la misma entrada, backtracking ha demostrado ser más eficiente en tiempo que branch and bound (para ulysses16 al menos), no obstante, ambos algoritmos son más lentos que programación dinámica, aunque consumen mucha menos memoria y los tres dan la solución óptima al problema.

Número de ciudades	Tiempo(Seg) backtracking	Tiempo(seg) Branch and Bound
3	8.358e-06	4.18e-05
4	2.0328e-05	7.1e-05
5	8.1754e-05	0.0002671
6	0.000712842	0.0009614
7	0.00212153	0.0038856
8	0.00620664	0.0105989
9	0.0408645	0.0552644
10	0.0407463	0.180636
11	0.266186	0.809554
12	1.09977	3.5446
13	5.46333	16.3346
14	35.473	86.1361