Algorítmica 2019/20

Práctica 5:

Backtracking y ramificación y poda

Índice:

Realizado por:

Jorge Medina Romero

Alberto Robles Hernández

Ahmed Brek Prieto

Mohammed Lahssaini Nouijah

1.1-. Problema del viajante de comercio Backtracking

1.1.1-.Planteamiento del problema

Nos hemos basado en un árbol, cada Nodo contiene:
Nodo{Número de la ciudad,
El coste acumulado de llegar hasta esa ciudad,
El camino para llegar hasta esta ciudad partiendo de la ciudad inicial}

La generación del árbol se hace en profundidad, es decir cada vez que generamos un hijo lo exploramos antes de generar el resto de los hijos.

Una vez llegamos a un nodo hoja actualizamos la cota superior, si el coste de dicho camino lo mejora; Cuando tenemos la cota superior, si estamos explorando un camino y en cualquier momento este sobrepasa el valor de la cota superior, se poda y deja de explorarse.

Pseudocódigo del algoritmo

```
FUNCIÓN TSP_BACKTRACKING(NodoActual, matrizAdyacencia, cotaSuperior, NodoSolución)
        IF(matrizAdyacencia.size() == NodoActual.camino.size())
                IF(cotaSuperior >= NodoActual.coste)
                        cotaSuperior = actual.coste
                        NodoSolucion = NodoActual
                END
                RETURN
        END
        FOR(i=0; i<matrizAyacencia.size(); ++i)
                IF(No hemos visitado todavía la ciudad i)
                        hijo.ciudad = i;
                        hijo.camino = actual.camino;
                        hijo.push(i);
                      hijo.coste = actual.coste+distancias[actual.ciudad][hijo.ciudad]
                        IF(hijo.coste < cotaSuperior)
                                TSP_BACKTRACKING(hijol, matrizAdyacencia, cotaSuperior,
                                NodoSolución)
                        //ELSE
                                //PODAMOS ESTA RAMA
                        END
                END
        END
        RETURN
END
```

1.1.2-.Eficiencia teórica

En este algoritmo existe la posibilidad de podar, por lo que los tiempos de ejecución del algoritmo dependen en gran parte del conjunto de datos de entrada, y no tanto del tamaño del problema. Esto se debe a que según los valores de los datos, se ejecutarán unas podas u otras teniendo un mismo tamaño del problema.

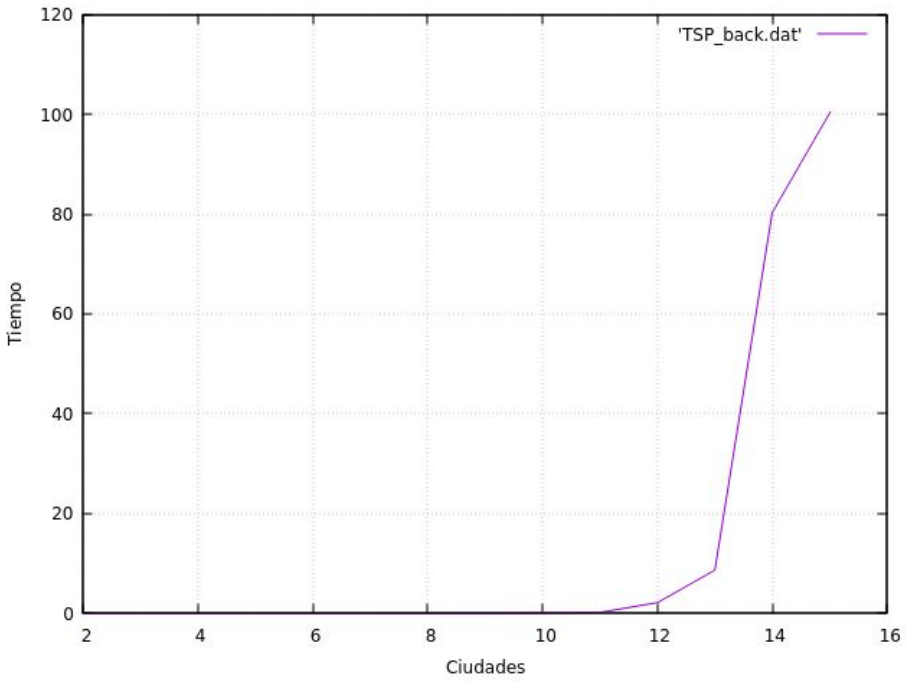
Sin embargo, consideremos el peor de los casos. Podemos asumir que este sería el caso en que no se ejecutara ninguna poda, y que la solución se hallara al explorar el último nodo del árbol, resultando así en una búsqueda en profundidad. En otras palabras, estaríamos estudiando el caso en que la poda no se podría ejecutar y por tanto no supondría ninguna mejora en eficiencia respecto del algoritmo en el que está basado (primero en profundidad), por lo que el orden de eficiencia (complejidad temporal) del algoritmo puede considerarse O(n!). Esto se debe a que el árbol se exploraría por completo, y por cada nodo que visitemos, deberemos expandir tantos hijos como ciudades restantes queden (se van decrementando de uno en uno por cada nivel del árbol, según las vamos visitando).

Sin embargo, cabe destacar que sí que se mantienen las ventajas que ofrece el recorrido de búsqueda en profundidad respecto a espacio, por lo que la complejidad en espacio no supone un problema frente a la complejidad en tiempo.

1.1.3-.Eficiencia empírica

Resultado tras ejecutar con ciudades aleatorias para analizar la eficiencia empírica. Ahora bien, sabemos que el tiempo final dependerá del orden en el que se analicen las ciudades, ya que se podrán podar soluciones y obtener así un tiempo final menor. Esto se aprecia por ejemplo, en el resultado de ejecutar el programa con 3 ciudades, al tardar 9e-06 segundos. Mientras que con 4 ciudades tarda 2.4e-05 segundos, es decir, aunque tengamos una ciudad más, ha tardado menos de la mitad que con una ciudad menos. Esto se debe a que se han podado nodos candidatos, ahorrando así tiempo de cómputo.

Número de Ciudades	Tiempo
2	1.7e-05
3	9e-06
4	2.4e-05
5	0.000118
6	0.000508
7	0.002673
8	0.013212
9	0.062844
10	0.31633
11	1.42818
12	1.99139
13	2.18284
14	8.70441
15	80.4611



1.1.5-.Caso de ejecución

```
Nombre de fichero:  ulysses16.tsp

Indice ciudad   Coordenadas (x,y)
        0       38.24   20.42
        1       39.57   26.15
        2       40.56   25.32
        3       36.26   23.12
        4       33.48   10.54
        5       37.56   12.19
        6       38.42   13.11

Numero de ciudades: 7

        *Matriz de Costes*

inf     5       5       3       10      8       7
5       inf     1       4       16      14      13
5       1       inf     4       16      13      12
3       4       4       inf     12      11      10
10      16      16      12      inf     4       5
8       14      13      11      4       inf     1
7       13      12      10      5       1       inf


Num Podas: 344
Num nodos extraidos: 234


Camino FINAL
0 --> 1 --> 2 --> 3 --> 4 --> 5 --> 6 --> 0

Ciudades en total = 7
Coste FINAL = 34
Tiempo de ejecucion: 0.001461
```

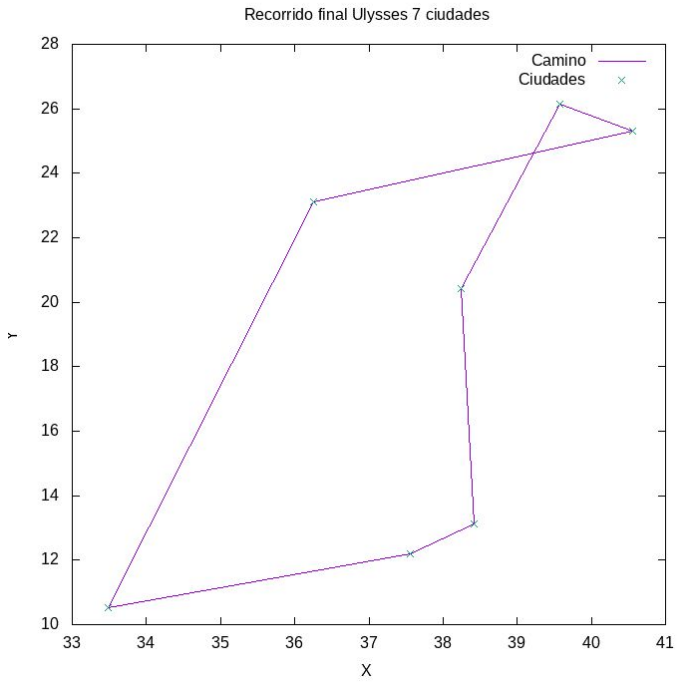
Analizamos el caso concreto de por ejemplo las siete primeras ciudades del fichero de datos ulysses16.tsp usando la estrategia de Backtracking “Vuelta atrás”.

Vemos las coordenadas de las siete ciudades, empezando por la ciudad 0 hasta la 6.

Tras eso, mostramos la matriz de costes asociada a esas ciudades y la distancia que hay entre cada una de ellas.

El camino resultante sería:
0→ 1→ 2→ 3→ 4→ 5→ 6→ 0
Con un coste total de 34.

* En la siguiente imágen se muestra el recorrido en 2D usando gnuplot.



Como vemos, el número de podas totales es de 344 y el total de nodos extraídos de un total de 234 nodos.

El tiempo de ejecución es de 0.0015 segundos.

1.2-. Problema del viajante de comercio Ramificación y poda

1.2.1-.Planteamiento del problema

Nos hemos basado en un árbol, cada Nodo contiene:
Nodo{Número de la ciudad,
El coste,
El camino para llegar hasta esta ciudad partiendo de la ciudad inicial,
La matriz de su padre reducida}

La generación del árbol se basa en el algoritmo de costo uniforme, es decir generamos todos los hijos antes de comenzar a explorarlos, seguimos la estrategia LC (Least Cost), es decir seleccionamos de entre todos los nodos de la frontera, el que nos genera un menor coste.

Una vez llegamos a un nodo hoja actualizamos la cota superior, si el coste de dicho camino lo mejora. Cuando tenemos la cota superior, si estamos explorando un camino y en cualquier momento este sobrepasa el valor de la cota superior, se poda y deja de explorarse.

Pseudocódigo del algoritmo

```
FUNCIÓN TSP_RAMIFICACIÓN_Y_PODA(ciudadActual,matrizDeAdyacencia)
        ColaDePrioridad<Nodo> Cola;
        actual = ciudadInicial;
        actual.coste = costeDeReducirLaMatriz(matrizDeAdyace)
        actual.camino.push(ciudadInicial)

        Cola.push(actual)

        WHILE(Cola no este vacia)
                Cola.pop();
                IF(CotaSuperior > actual.coste)
                        IF(actual.camino.size() == matrizDeAdyacencia.size())
                                CotaSuperior = actual.coste
                                Solucion = actual
                        ELSE
                                FOR (i in 0 to actual.numHijos)
                                        hijo = actual.generamosHijo(i)
```

```
                        hijo.coste = hijo.calcularCoste()
                        Cola.push(hijo)
                    END
                END
            //ELSE
                //PODAMOS ESTA RAMA
            END
            IF(cola no esta vacia)
                actual = cola.Top()
            END
        END
        DEVOLVER(Solucion)
END
```

1.2.2-.Eficiencia teórica

La eficiencia de este algoritmo es muy similar al del algoritmo basado en Backtracking, ya que su funcionamiento es bastante parecido. De esta manera, comparten la capacidad de realizar podas y ahorrar una gran cantidad de cálculos cuando predice que un camino no va a contener la solución según los cálculos ya realizados.

Nuevamente, la eficiencia de la ejecución del algoritmo puede variar enormemente según los valores de los datos de entrada aunque conservemos el mismo tamaño de problema. De esta manera, debemos volver a considerar el peor de los casos, en el que no se realizaría ninguna poda. Es entonces en este caso donde se recorrería el árbol por completo, por lo que la eficiencia en el peor de los casos sería O(n!).

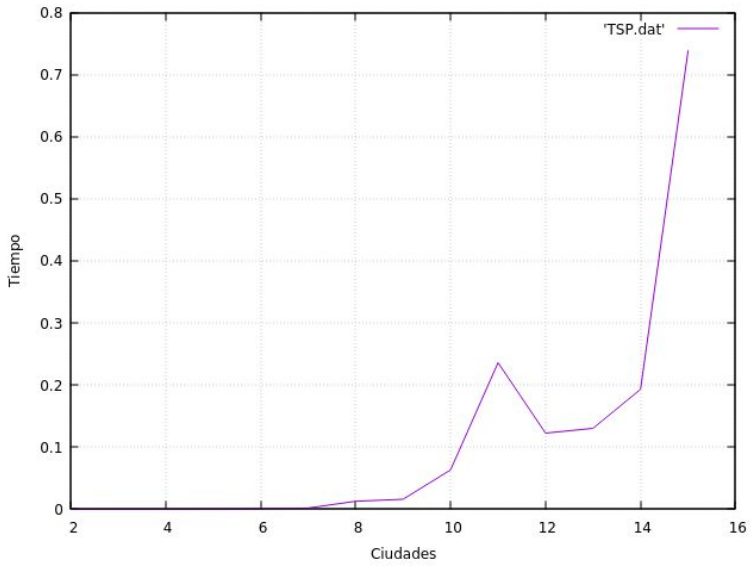
La diferencia respecto al otro algoritmo reside en la complejidad espacial, que sería superior a la del otro algoritmo, pues tendría muchos más nodos abiertos por el simple hecho de que basa su recorrido en el costo uniforme, incrementándose así el tamaño de la frontera de nodos abiertos.

A pesar de todo esto, cabe destacar que la búsqueda de este algoritmo va más guiada para encontrar antes una solución óptima, por basarse en costo uniforme.

1.2.3-.Eficiencia empírica

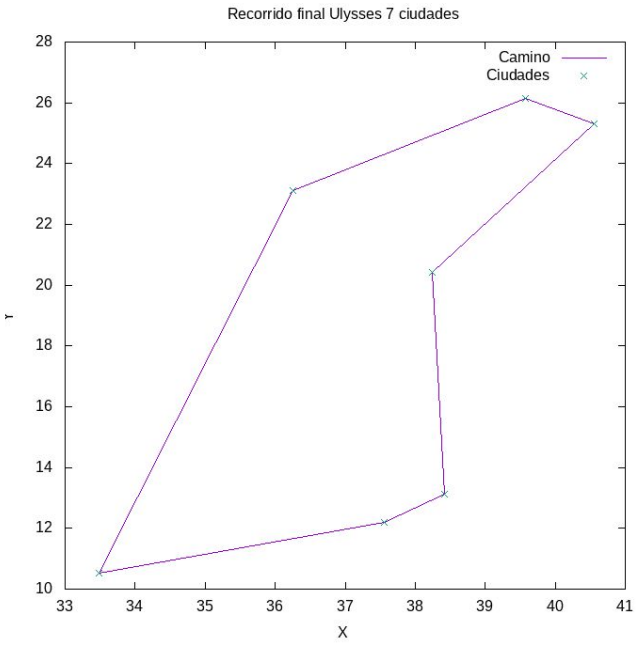
Resultado tras ejecutar con ciudades aleatorias para analizar la eficiencia empírica. Ahora bien, sabemos que el tiempo final dependerá del orden en el que se analicen las ciudades, al igual que en backtracking ya que se podrán podar soluciones y obtener así un tiempo final menor. Esto se aprecia por ejemplo, en el resultado de ejecutar el programa con 11 ciudades, al tardar 0.236045 segundos. Mientras que con 12 ciudades tarda 0.122531 segundos, es decir, aproximadamente la mitad. Esto se debe a que se habrán podado nodos candidatos, ahorrando así tiempo de cómputo.

Número de Ciudades	Tiempo
2	2.4e-05
3	0.000125
4	0.000217
5	0.000439
6	0.000948
7	0.001957
8	0.012765
9	0.016125
10	0.063357
11	0.236045
12	0.122531
13	0.130328
14	0.193003
15	0.738045



1.2.5-.Caso de ejecución

```
Nombre de fichero:  ulysses16.tsp

Indice ciudad   Coordenadas (x,y)
        0       38.24   20.42
        1       39.57   26.15
        2       40.56   25.32
        3       36.26   23.12
        4       33.48   10.54
        5       37.56   12.19
        6       38.42   13.11

Numero de ciudades: 7

        *Matriz de Costes*

inf     5       5       3       10      8       7
5       inf     1       4       16      14      13
5       1       inf     4       16      13      12
3       4       4       inf     12      11      10
10      16      16      12      inf     4       5
8       14      13      11      4       inf     1
7       13      12      10      5       1       inf


Num Podas: 215
Num nodos extraidos: 118
Tam maximo de cola con prioridad de nodos vivos: 118


Camino FINAL
0 --> 6 --> 5 --> 4 --> 3 --> 1 --> 2 --> 0


Ciudades en total = 7
Coste FINAL = 34
Tiempo de ejecucion: 0.011902
```



Analizamos el caso concreto de por ejemplo las siete primeras ciudades del fichero de datos ulysses16.tsp usando la estrategia ahora de Branch and Bound “Ramificación y poda”.

Vemos las coordenadas de las siete ciudades, empezando por la ciudad 0 hasta la 6.

Tras eso, mostramos la matriz de costes asociada a esas ciudades y la distancia que hay entre cada una de ellas.

El camino resultante sería:
0→ 6→ 5→ 4→ 3→ 1→ 2→ 0
Con un coste total de 34. Al igual que teníamos calculado con Backtracking.

*En la siguiente imágen se muestra el recorrido en 2D usando gnuplot.

En este caso, el camino varía en la ciudad 1→ 2→ 0 , que en Backtracking era 2 → 1 → 0, pero el coste se mantiene constante, así que la solución es válida. Esto se debe a que cada estrategia recorre el árbol de forma diferente.

Como vemos, el número de podas totales es de 215 (menos podas que en Backtracking) y el total de nodos extraídos es de 118 nodos (también menor que en Backtracking).
El tamaño máximo de la cola con prioridad de nodos vivos es de 118.
Por último, el tiempo de ejecución es de 0.0012 segundos.

1.3-.Comparación con las distintas aproximaciones TSP

	Eficiencia	Optimalidad
Greedy	$O(n^2)$	No
Programación dinámica	$O(n^2 * 2^n)$	Sí
Backtracking	$O(n!)$	Sí
Branch & Bound	$O(n!)$	Sí

* Respecto a la complejidad espacial, el peor es el algoritmo basado en PD, ya que ocupa mucho espacio en la pila, dada su recursividad y los datos que almacena. Los algoritmos Greedy serían los mejores en este aspecto, ya que no emplean recursividad y el espacio que necesitan por iteración es limitado.

	Camino	Costo	Tiempo (con n=10)
TSP Backtracking	0 --> 5 --> 1 --> 8 --> 2 --> 6 --> 9 --> 4 --> 3 --> 7 --> 0	282	0.382811
TSP Branch & Bound	0 --> 7 --> 3 --> 4 --> 9 --> 6 --> 2 --> 8 --> 1 --> 5 --> 0	282	0.031713
TSP Greedy	0 --> 5 --> 7 --> 1 --> 8 --> 2 --> 4 --> 3 --> 6 --> 9 --> 0	358	0.000161
TSP Programación dinámica	0 --> 5 --> 1 --> 8 --> 2 --> 6 --> 9 --> 4 --> 3 --> 7 --> 0	282	0.177182

→ Podemos observar que aunque la eficiencia teórica (en el peor de los casos) de los algoritmos de poda sean mucho peores que el resto de algoritmos, a la hora de la práctica sus tiempos de ejecución son, por lo general, mucho menores. Esto se debe a que no suele darse el caso de que se ejecute el peor de los casos, y la realización de las podas decrementan enormemente el tiempo de cálculo. Por tanto, a pesar de su aparentemente mala eficiencia teórica, pueden ser preferibles que el resto de algoritmos.

→ Además si tuviésemos que elegir entre Backtracking o Branch & Bound sería preferible elegir la estrategia Branch & Bound ya que, aunque ambas tienen el mismo nivel de eficiencia, con Branch and Bound el tiempo de ejecución promedio suele ser menor que el obtenido por Backtracking.

2-. Problema 4 - ITV con Backtracking

2.1-. Planteamiento del problema

Para el problema de ITV con backtracking hemos planteado un árbol de búsqueda, en el que, en cada nivel, se decide una cola para el coche coche[nivel]. Es decir, cada nodo tiene M hijos, siendo M el número de estaciones de la ITV de las que disponemos, lo que nos permite considerar todas las posibles asignaciones de un coche a cada línea.

Una vez exploramos una rama completa, hasta llegar al nodo hoja, es posible actualizar el valor de la cota superior (si el tiempo final de las colas es menor que la cota superior). Cada vez que vayamos a descender por un nodo, comparamos su coste actual con el de la cota superior. Si la cota superior es menor que el valor actual, podemos afirmar que esa subdistribución de los coches en las colas no nos va a llevar a la solución óptima, por lo que podemos podar esa rama.

Pseudocódigo del algoritmo

```
FUNCIÓN DISTRIBUCIONCOCHES(coches, padre, cotaSuperior, solucion)
        IF(padre.size() == coches.size())  // Si hemos distribuido todos los coches
                IF(cotaSuperior >= padre.tiempoGlobal)
                        cotaSuperior = padre.tiempoGlobal
                        solucion = padre
                END
                RETURN
        END
        FOR(i=0; i<padre.indicesColas.size(); ++i)
                hijo = aniadirCocheACola(coche, cola, tiempoInspeccion)

                IF(hijo.tiempoAcumulado < cotaSuperior)
                        DISTRIBUCIONCOCHES(coches, hijo, cotaSuperior, solucion)
                //ELSE
                        //PODAMOS ESTA RAMA
                END
        END
        RETURN
END
```

2.2-. Eficiencia teórica

Al ser un algoritmo basado en Backtracking, podemos considerar la explicación del algoritmo anterior basado en esta técnica. Es decir, en resumen, en el peor de los casos se recorrería todo el árbol. Sin embargo, hay una diferencia notable entre el problema anterior y éste: la forma del árbol. En el caso anterior, cada nodo tendrá un número de hijos dependiendo del nivel en que se encuentre (y este factor de ramificación se iba decrementando según se bajaba de nivel, ya que cada vez teníamos menos opciones para considerar). En este caso es distinto, ya que las posibilidades siempre son las mismas: siempre podemos llevar un coche a cualquiera de las líneas de la estación. De esta forma, el factor de ramificación es constante para todo el árbol, siendo este igual al número de líneas que consideremos en cada problema.

Teniendo todo esto en cuenta, consideramos las dos entradas que definen el tamaño del problema: número de coches (C) y número de líneas (N). De ahí, concluimos que el tamaño del árbol y por tanto el orden de eficiencia es $O(C^N)$. Al basarse en backtracking, es bastante eficiente en espacio, por lo que, de nuevo, el principal problema es el tiempo, no el espacio.

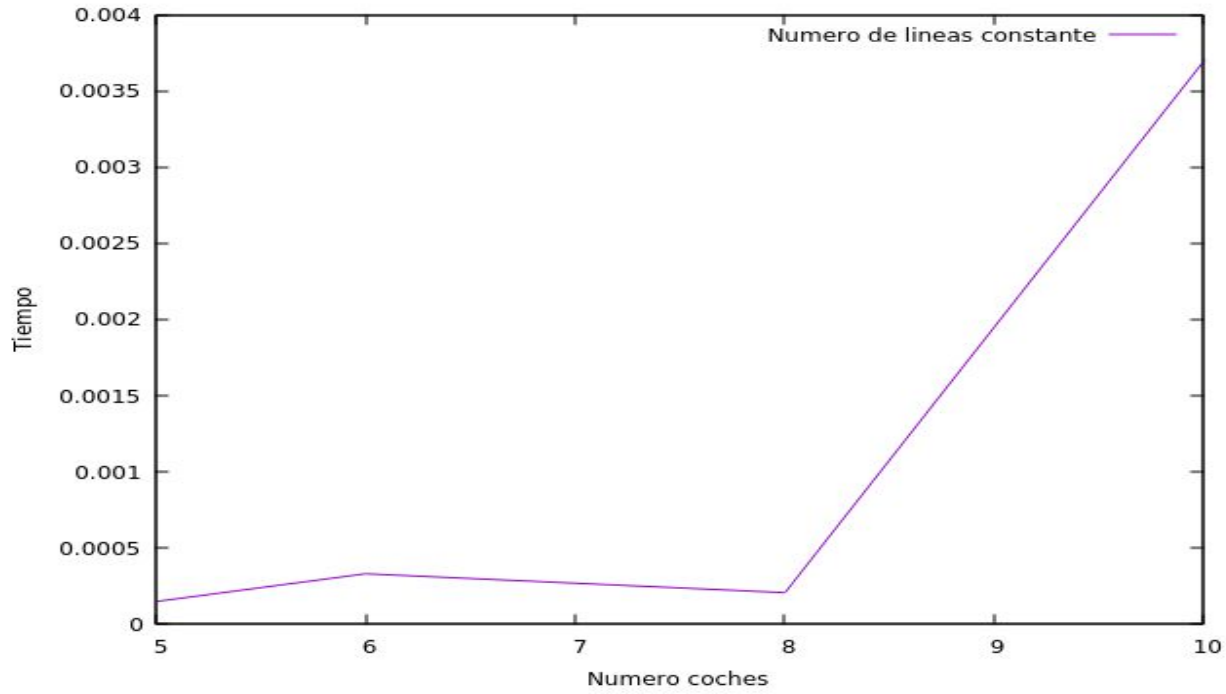
2.3-. Eficiencia empírica

Num líneas	Num coches	Tiempo
2	5	0.00015
2	6	0.000332
2	8	0.000208
2	10	0.0037
3	10	0.02212
4	10	0.014021
5	10	0.053597

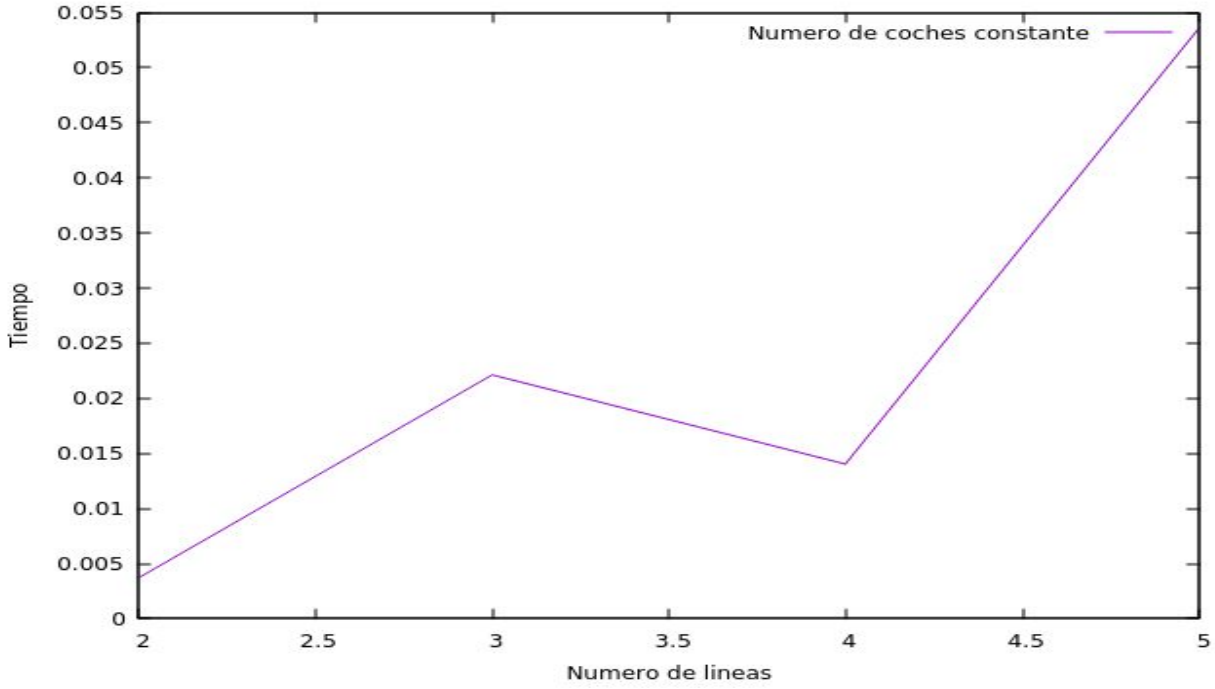
* Hemos hecho distintas ejecuciones, unas variando el número de coches y otras, el número de líneas, ya que ambas son las entradas del problema.

Al igual que antes, no tiene demasiado sentido calcular la eficiencia empírica con algunos conjuntos de datos concretos, ya que el número de podas posibles a realizar dependerá de la entrada de datos, y esta puede variar.

Gráfica que ilustra el comportamiento del programa variando el número de coches y dejando constante el número de líneas de inspección (en concreto, 2 líneas de inspección).



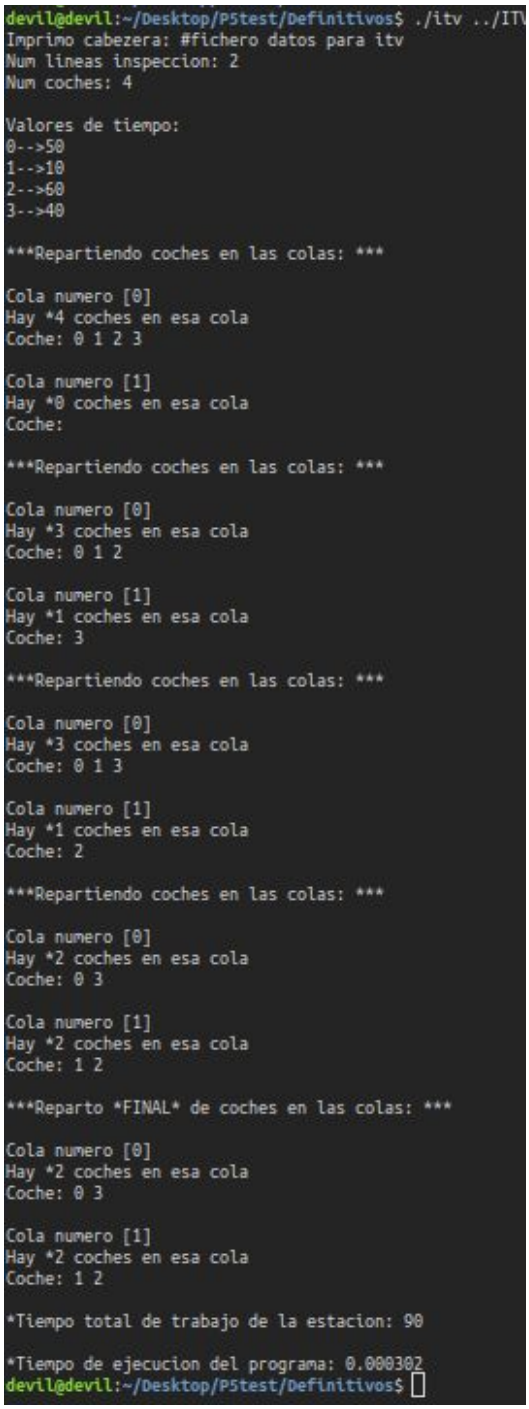
En esta otra gráfica variamos el número de líneas de inspección y dejamos constante el número de coches (en concreto, 10 coches)

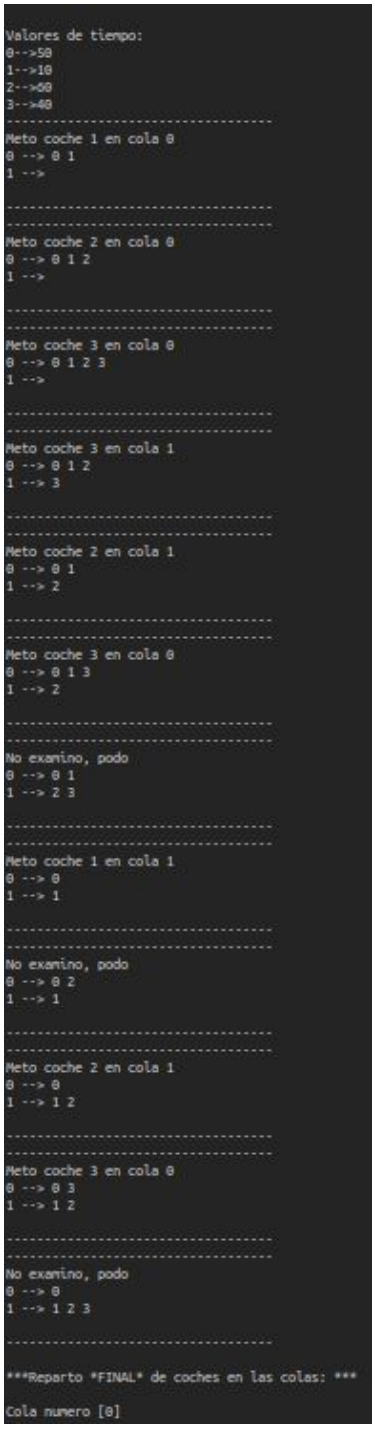


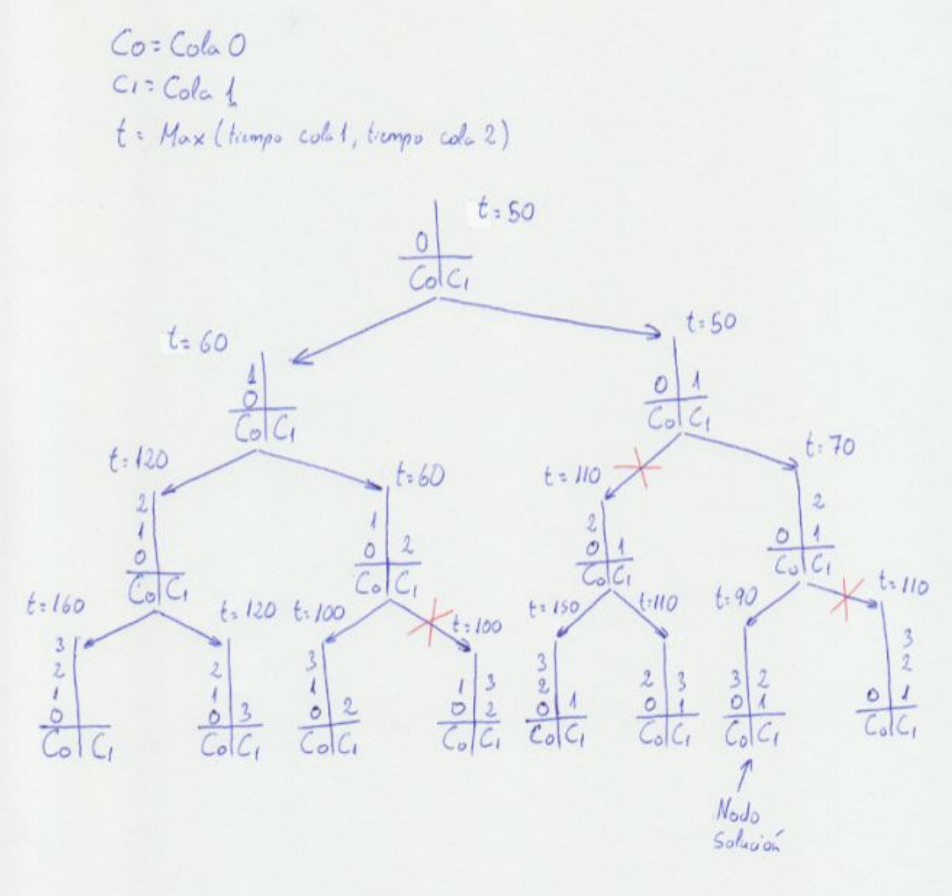
2.5-.Caso de ejecución

Analizamos el caso concreto de por ejemplo tener 2 líneas de inspección y 4 vehículos.

En la imágen de la izquierda vemos cómo va cambiando la cola de nodos vivos y en la de la derecha vemos cómo se va subiendo y bajando en cada uno de los niveles según vamos explorando el árbol.







Coche 0 - t=50
Coche 1 - t=10
Coche 2 - t=60
Coche 3 - t=40

En las cruces rojas que cortan las distintas ramas se realiza una poda, debido a que cotaSuperior <= t