

Doble Grado en Ingeniería Informática y Matemáticas

Algorítmica

**Práctica 4 - Exploración de Grafos**

David Kessler Martínez

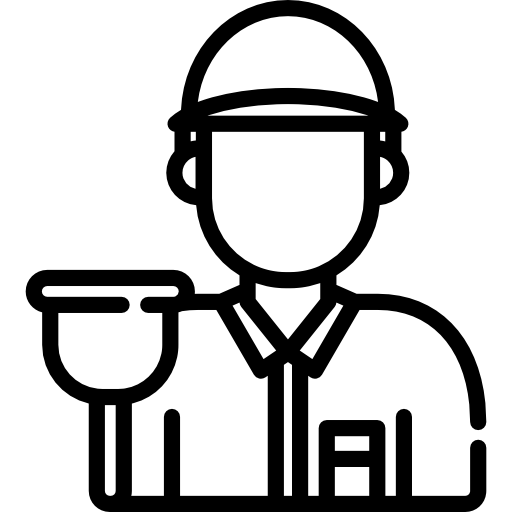
[dkesslerm03@correo.ugr.es](mailto:dkesslerm03@correo.ugr.es)

Santiago López Cerro

[santiloce@correo.ugr.es](mailto:santiloce@correo.ugr.es)

Antonio Javier Rodríguez Romero

[antoniojrr@correo.ugr.es](mailto:antoniojrr@correo.ugr.es)



Granada, curso 2022/2023

Índice

**0**[**. Introducción**](#_y1jy1bvhlgeb) 3

[0.1. Objetivos](#_xk1xpcg6ebty) 3

[0.2. Entorno y autores](#_xk1xpcg6ebty) 3

[0.2.1. Autores](#_xk1xpcg6ebty) 3

[0.2.2. Entorno de análisis](#_mjk3082qaow3) 3

[Santiago López Cerro](#_t4s09tmx7yko) 3

[David Kessler Martínez](#_nqrwkvdjevuo) 3

[Antonio Javier Rodríguez Romero](#_3hb0b4ygmxff) 4

[0.2.3. Obtención de tiempos](#_ofxykke5mnwh) 4

[**1.P4: Electricista con penalizaciones**](#_y1jy1bvhlgeb) **5**

[1.1. Análisis del problema](#_hghneixmciq9) 5

[1.2. Algoritmo ***BackTracking***](#_hghneixmciq9) 5

[Justificación de validez](#_5aaif9e6ienc) 8

[1.2.1. Análisis de eficiencia](#_a6ex9929fei9) 9

[1.2.1.1. Análisis teórico 9](#_a6ex9929fei9)

[1.2.1.2. Análisis empírico 9](#_a6ex9929fei9)

[1.2.1.3. Análisis híbrido](#_odur49ghdohn) 10

[1.2.2. Estudio de cotas](#_a6ex9929fei9) 12

[1.3. Algoritmo ***Branch and Bound***](#_hghneixmciq9) 13

[Justificación de validez](#_5aaif9e6ienc) 14

[1.2.1. Análisis de eficiencia](#_a6ex9929fei9) 14

[1.2.1.1. Análisis teórico](#_a6ex9929fei9) 14

[1.2.1.2. Análisis empírico](#_a6ex9929fei9) 15

[1.2.1.3. Análisis híbrido](#_odur49ghdohn) 16

[1.2.2. Estudio de cotas](#_a6ex9929fei9) 17

[1.4. Comparativa de algoritmos](#_hghneixmciq9) 18

**2. Conclusiones****19**

0. introducción

0.1. Objetivos

Esta práctica pretende resaltar la utilidad de las técnicas de creación de algoritmos ***Backtracking*** y ***Branch and Bound*** en casos en los que tratemos con recorridos en grafos. Además, se observará cuál de estos es más eficaz resolviendo un mismo problema y, en el caso de que exista diferencia entre los resultados obtenidos con estos, cúal de ellos consigue uno mejor.

0.2. Autores y Entorno de trabajo

### 0.2.1. Autores

David Kessler Martínez - 33%

Antonio Javier Rodríguez Romero - 33%

Santiago López Cerro - 33%

### 0.2.2. Entorno de análisis

Estos los distinto entornos de cada uno de los compañeros en donde se ha realizado el estudio de los algoritmos:

#### Santiago López Cerro

* Dispositivo: HP Pavilion x360 Convertible 14-dw1xxx
* Procesador: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz
* RAM instalada: 8,00 GB (7,65 GB usable)
* Arquitectura: Sistema operativo de 64 bits, procesador basado en x64
* CPU(s): 4
* Hilo de procesamiento por núcleo: 1
* Socket(s): 1
* Núcleos por socket: 4

Especificaciones de Windows:

* Sistema Operativo: Windows 10 Home

Acerca de mi maquina virtual:

* Memoria de base: 2048 MB
* Procesadores: 4
* Orden de arranque: Disquete, Óptica, Disco Dura

#### David Kessler Martínez

Dispositivo: Lenovo IdeaPad 5 15ALC05

Procesador: AMD Ryzen 5 5500U with Radeon Graphics

RAM: 16 GB

Arquitectura: x86\_64

Sistema operativo: Ubuntu 22

CPU(s):12

Hilo de procesamiento por núcleo: 2

Socket(s): 1

Núcleos por socket: 6

#### Antonio Rodríguez Romero

Dispositivo: ASUS TUF Dash F15

Procesador: 11th Gen Intel(R) Core(TM) i7-11370H @ 3.30 GHz

RAM: 16GB

Arquitectura: x86\_64

Sistema operativo: Ubunto 22

CPU(s): 8

Hilo de procesamiento por núcleo: 2

Socket(s): 1

Núcleos por socket: 4

En cuanto a la compilación, hemos utilizado el compilador g++ con nivel de optimización 1.

### 

### 0.2.3. Obtención de tiempos

En lo relativo al método de medición de tiempo hemos decidido optar por el uso de la biblioteca ctime, haciendo uso de la función clock() antes y después de la ejecución del método que estudiemos, de manera que encontremos el tiempo de ejecución de este al restar al instante de después de esta el tiempo antes. Además, para mayor estabilidad y mejor representación, el algoritmo se ejecutará un total de 5 veces de forma que el resultado será la media de estos tiempos.

1. P4: Electricista con penalizaciones
   1. Análisis del problema

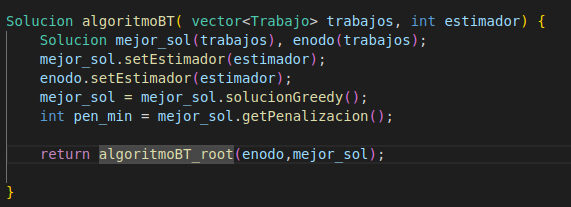
Nuestro problema, **Electricista con penalizaciones**, nos presenta una situación en la que tenemos pendientes **n trabajos**, de los cuales conocemos la duración de estos, es decir, lo que tardaríamos en terminarlos, el plazo máximo que disponemos para terminar cada uno de ellos y la penalización monetaria que sufriremos si no cumplimos este último.

Las dos heurísticas que creemos deberán establecer el orden de realización de trabajos para conseguir una penalización monetaria mínima. Así, nuestro vector X de soluciones va a almacenar el orden en que se ejecuta el trabajo i-ésimo.

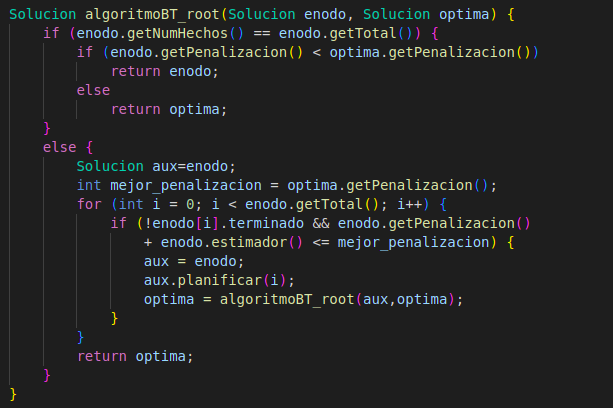
Tendremos que tener en cuenta tanto la prioridad de estos según su penalización como su duración. Primero, debemos pensar en la representación que utilizamos internamente para guardar estos datos. Como cada trabajo viene caracterizado por tres valores (duración, plazo y penalización), utilizaremos un struct Trabajo para almacenarlos. Guardaremos todos los Trabajos en un vector y, para las soluciones, en el caso del algoritmo Branch and Bound, utilizaremos una cola con prioridad.

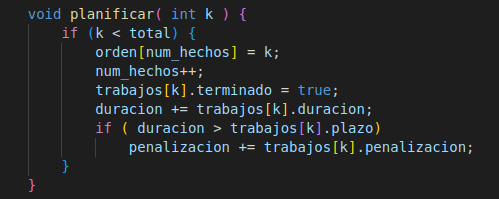
* 1. Algoritmo *Backtracking*

Cómo hemos mencionado antes, usamos un struct Trabajo. Además de los tres valores que caracterizan a cada trabajo, hemos incluido también un bool terminado para indicar si el trabajo ya se ha realizado o no. Por otra parte, utilizamos un TDA Solución que incluye diferentes constructores, un método que ejecuta un algoritmo Greedy para la primera cota global, un método estimador (que según el valor introducido por el usuario, escoge entre las distintas maneras de calcular la cota local) y la sobrecarga de distintos operadores para nuestro TDA Solución, entre otros. Se puede consultar el resto del contenido de esta clase en cualquiera de los archivos bt.cpp o bb.cpp.

Para resolver nuestro problema utilizando un algoritmo basado en Backtracking, hemos implementado la siguiente función en C++: 

Esta función llama a la función algoritmoBT\_root según el enodo actual, la cual tiene la siguiente estructura:

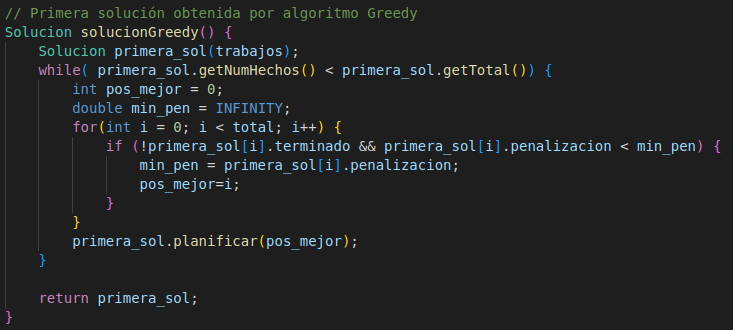


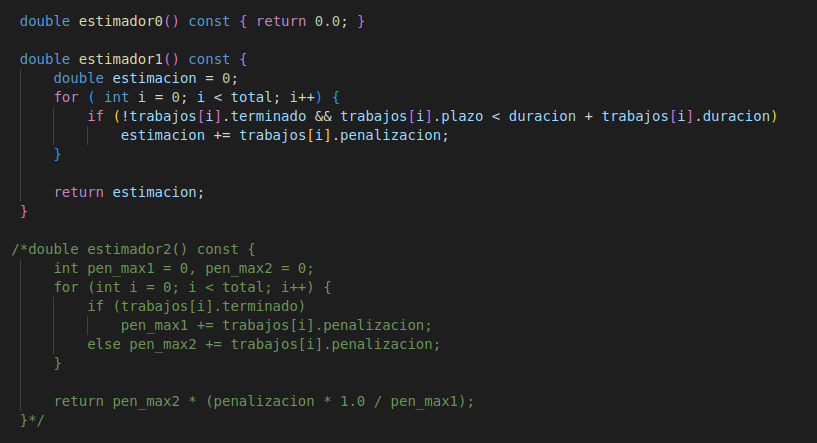
Vemos que esta función comprueba si tenemos una solución completa, en cuyo caso se devuelve la mejor entre la recién creada y la solución óptima. En caso contrario, se recorren todos los trabajos no terminados de la solución parcial, y si se cumple que la penalización de la solución junto con la cota local elegida es menor o igual que la mejor penalización obtenida hasta ahora, entonces se llama al método planificar y se vuelve a llamar a la función de manera recursiva. El método planificar es la siguiente:

el cual actualiza la Solución según el nuevo Trabajo añadido.

La solución es el atributo vector<int> orden, que cumple la función del vector de soluciones X, es decir, almacenar el orden en que se realizan los distintos trabajos.

Las restricciones explícitas serán, directamente, que no se asigne a ningún trabajo un orden mayor al número de trabajos totales que tenemos. Los distintos valores que podrá tomar la posición i-ésima del vector orden será 1, 2, …, n, siendo n el número total de trabajos. Las restricciones implícitas serán orden[i] != orden[j] para todo i distinto de j, ya que dos trabajos no pueden ser elegidos con un mismo orden.

Las cotas que hemos escogido son las siguientes. La primera cota que se calcula se hace mediante un algoritmo Greedy, implementado de la siguiente manera: 

Esta cota será la cota global, hasta que sea actualizada por la cota local de una hoja, que será menor que la cota global, al tratarse de un problema de minimización. Las cotas locales que se han considerado han sido las siguientes: 

+

En cada caso, el valor de la cota local es la suma de la función estimador escogida más la penalización que se tiene en la solución hasta el momento.

En el caso de estimador0, no se suma nada, y la cota local es únicamente el valor de la penalización acumulada hasta el momento. Aunque se trata de una cota local correcta y que provocará que se corten algunas ramas del árbol de estados, el siguiente estimador la ajustará mejor y de una manera segura también.

En el caso de estimador1, al valor de la penalización acumulada se le suma las penalizaciones de los trabajos no terminados cuyo plazo sea menor que el tiempo que ya llevamos acumulado trabajando, de forma que se tendrán en cuenta las penalizaciones restantes que seguro que vayamos a tener.

Pensamos en una tercera manera de calcular la cota local, para la cual implementamos el estimador2, pero no conseguimos ultimar la idea de manera que proporciona una solución válida de manera consistente. Esta estaba inspirada en la tercera cota del viajante de comercio (especificada en las transparencias, página 120), queríamos sumar al valor de la penalización acumulada un porcentaje de las penalizaciones de los trabajos restantes, siendo este el mismo que el de penalizaciones acumuladas sobre las máximas posibles. Sin embargo, este estimador, aunque en un principio pudiera parecer válido, como la acumulación de estas no es un proceso uniforme conforme realizamos la planificación, sino que es imprevisible, este estimador muchas veces provocaría que se cortaran ramas son válidas.

Más adelante se estudiará cuál de las dos cotas locales es más efectiva a la hora de encontrar la mejor solución.

### Validez de las cotas

El objetivo de la creación de los estimadores es aproximar de la manera más exacta posible, a partir de un nodo no hoja del árbol, el valor final de la penalización de la rama óptima que cuelga por debajo de este. Lo óptimo sería saber exactamente este peso, sin embargo, para ello tendríamos que estudiar uno a uno cada uno de los caminos de forma que la eficiencia del algoritmo disminuye exponencialmente.

Por lo tanto, con las cotas locales, siempre intentaremos conseguir un valor que se acerque a este peso final lo máximo posible pero necesariamente sin superarlo, ya que esto podría provocar no recorrer ramas que pudieran ser las óptimas.

En el primer estimador presentado, vemos como la cota local la obtendremos a partir de las penalizaciones ya acumuladas hasta el momento, de forma que el peso final de la rama será ese mismo o mayor, debido a que se sume alguna más, luego se trata de una estimación correcta.

En el segundo caso, a parte de las penalizaciones previas, se obtendrán los trabajos no realizados cuyo plazo ya ha cumplido, debido a que sus penalizaciones serán sumadas inevitablemente en algún momento, para sumar estas a las ya acumuladas. De esta forma el resultado de la suma será siempre igual a la final o menor, en el caso de que al añadir más trabajos, haya otros cuyo plazo expire.

# Análisis de eficiencia

## 1.2.1.1. Análisis teórico

En nuestro código es el método *algoritmoBT()* donde se implementa la técnica de búsqueda exhaustiva mediante el algoritmo de vuelta atrás (*backtracking*). A continuación, realizaremos su estudio teórico de la eficiencia del método:

- La función ***solucionGreedy()***: tiene una eficiencia de **O(n²)**, donde n es el número de trabajos. Esto se debe a que se realiza un bucle interno que busca el trabajo con la menor penalización en cada iteración, y este bucle se repite n veces, siendo por tanto la concatenación de ambos bucles lo que nos produce una eficiencia cuadrática.

- La función ***estimador1()***: tiene una eficiencia de **O(n)**, ya que realiza un bucle que recorre todos los trabajos para calcular una estimación basada en las penalizaciones y plazos de los trabajos no terminados. En el caso de *estimador0()*, es de orden O(1), ya que se trata del caso por defecto, donde se devuelve el valor 0.0 .

- La función ***algoritmoBT\_root()***: es recursiva y utiliza el algoritmo de vuelta atrás para explorar todas las posibles combinaciones de planificación de los trabajos. En el peor caso, el número de llamadas recursivas es 2^n, donde n es el número de trabajos. En cada llamada recursiva, se realiza un bucle de complejidad O(n) para verificar las condiciones y realizar las planificaciones. Por lo tanto, la complejidad de tiempo de `algoritmoBT\_root` es exponencial, **O()**.

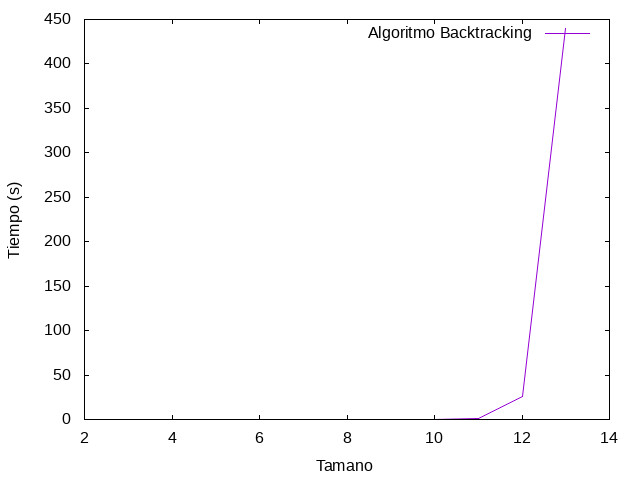
- En la función principal ***algoritmoBT()***, se realiza una llamada inicial a *solucionGreedy()* que tiene una complejidad de tiempo de O(n²). Luego, se llama a *algoritmoBT\_root()* que tiene una complejidad de tiempo exponencial **O()**. Por tanto, tratándose de llamadas separadas, la complejidad sería de **O( +** n²**) ∈**  **O().**

En resumen, la eficiencia del método *algoritmoBT()* es exponencial, **O()**.

## 1.2.1.2. Análisis empírico

| ***Nº de trabajos*** | ***Tiempos (s)*** |
| --- | --- |
| 3 | 3.2e-06 |
| 4 | 8.8e-06 |
| 5 | 2.34e-05 |
| 6 | 0.0001812 |
| 7 | 0.00082 |
| 8 | 0.0063436 |
| 9 | 0.0380658 |
| 10 | 0.229025 |
| 11 | 0.68171 |
| 12 | 25.5368 |
| 13 | 440.311 |

Representación gráfica de los resultados obtenidos:

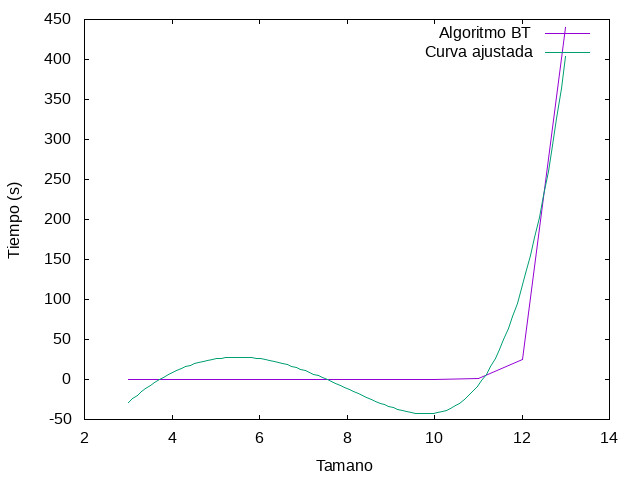


## 1.2.1.3. Análisis híbrido

Como hemos visto en el estudio teórico se trata de un algoritmo del modo 2n \* n por lo que consideraremos la siguiente ecuación como nuestra ecuación de ajuste:

A través del software *gnuplot*, deducimos por regresión mediante mínimos cuadrados que las constantes ocultas serán

Representación gráfica:

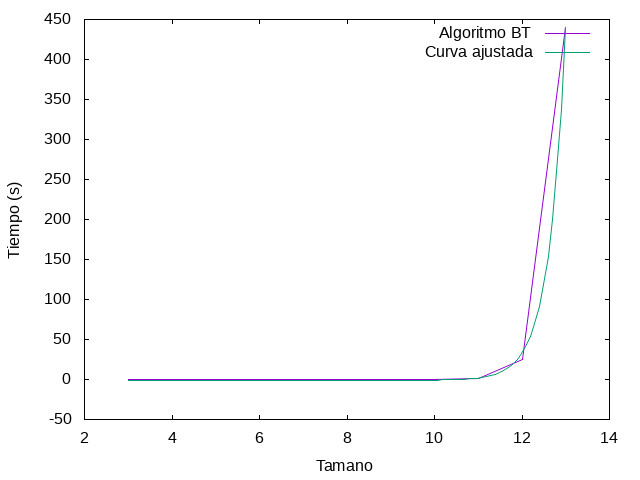


Como vemos el estudio teórico no consigue reflejar del todo a la perfección las variaciones que pueden proporcionar los datos de entrada de caso. A pesar de esto la tendencia exponencial vemos que es tal y como esperábamos teóricamente.

Mediante un ajuste con una función factorial, observamos que se trata de un ajuste mejor. Es por tanto que deducimos que la entrada de los datos y su arbitrariedad acaba determinando unos resultados más cercanos al caso peor, que es del tipo factorial.

La función que he implementado es la siguiente:

Finalmente los valores de las constantes ocultas son los siguientes:

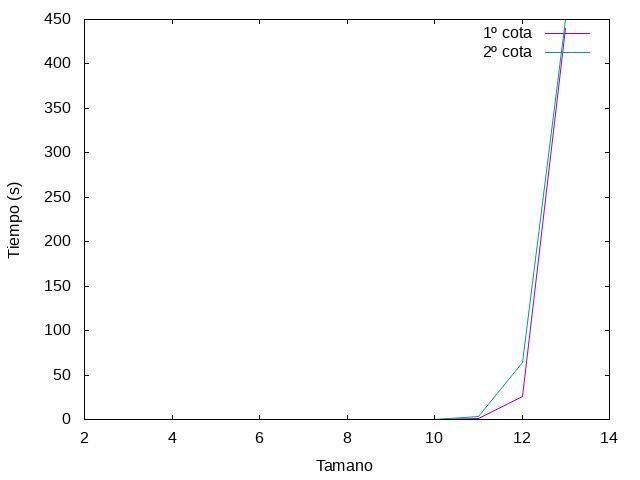


# 1.2.2. Estudio de cotas

Para finalizar el estudio de la técnica ***Backtracking*** usada en la implementación del algoritmo de nuestro ejercicio realizaremos una comparativa entre las dos cotas que hemos empleado:

1º. **Cota 0 (*estimador0()*)**→ Solo tiene en cuenta las penalizaciones acumuladas hasta el momento

2º. **Cota 1 (*estimador1()*)** → Realiza una previsión en base a los plazos ya cumplidos.



Apreciamos en esta gráfica comparativa que la elección de una cota frente a otra no supone una mejora tan significativa en el eficiencia del algoritmo como sí podría serlo en el algoritmo Backtracking.

* 1. Algoritmo *Branch and Bound*

La estructura del struct Trabajo, el TDA Solución y todas los métodos y funciones asociadas son iguales que para el algoritmo Backtracking, al igual que las cotas y sus cálculos. Por no repetir información, en este apartado únicamente estudiaremos el código del algoritmo Branch and Bound. Es el siguiente:



La diferencia principal con el algoritmo anterior es que ahora no nos limitamos a un solo enodo, es decir, cuando vemos que una solución es factible la almacenamos en una cola con prioridad, priority\_queue<Solution> Q, y seguimos recorriendo el mismo nivel del árbol antes de pasar al siguiente. Una vez tenemos todas las posibles soluciones almacenadas en la cola con prioridad (ya ordenadas, utilizando como función de prioridad la sobrecarga del operador <, mostrada a continuación), cogemos la más prometedora de todas ellas, es decir, la primera de la cola con prioridad, y la expandimos, repitiendo este proceso hasta llegar a una solución completa. Si la solución a la que se llega expandiendo el enodo más prometedor es mejor que la que ya teníamos, esta se actualiza, y se pasa al siguiente elemento más prometedor, repitiendo el proceso.

Las restricciones explícitas e implícitas son las mismas que para el algoritmo Backtracking, y las distintas funciones de cota, así como la elección de la primera cota a través del algoritmo Greedy, también lo son.

### Justificación de validez

Este algoritmo recorrerá el árbol de estados partiendo desde un estado nulo o vacío hacia abajo, de manera que estudiará todas los estados factibles del árbol por niveles. Como por las cotas locales establecidas, sabemos que nunca se podará una rama de este que pudiera contener las solución óptima, sino que solo se cortarán aquellas de las que estemos seguros que no la contienen, podemos deducir que se encontrará la secuencia de trabajos buscada.

# Análisis de eficiencia

## 1.3.1.1. Análisis teórico

Nuestro algoritmo de ramificación y poda utiliza una estructura de datos de cola de prioridad (*priority queue*) para almacenar las soluciones parciales y procesarlas posteriormente en orden de menor penalización. Esta cola es implementada mediante un montículo binario que conlleva un tiempo de inserción y extracción de elementos de ***O(log n)***, donde n es el número de elementos de cola.

Sobre la función de límite inferior esta se implementa mediante el método *estimador()*, que proporciona una estimación del costo mínimo de una solución parcial.

El algoritmo utiliza una solución inicial proporcionada mediante el algoritmo *Greedy* que ordena los trabajos por su cociente penalización/duración. Dicho algoritmo tiene una complejidad de ***O(n)***, donde n es el número de trabajos. En cada iteración del bucle principal, se extrae la solución parcial de menor penalización de la cola de prioridad. Si la solución parcial ya contiene todos los trabajos verificamos entonces si es mejor que la solución encontrada hasta ahora, lo que conlleva un costo constante de ***O(1)***.

En caso contrario, de que la solución parcial no contuviese todos los trabajos, se generan nuevas soluciones añadiendo un trabajo no terminado a la solución parcial actual. Esto se realiza mediante un bucle que itera sobre los trabajos no terminados y genera una nueva solución añadiendo cada uno de ellos a la solución parcial. En el peor de los casos, el bucle ha de iterar sobre todos los trabajos no terminados, lo que tiene una complejidad de ***O(n)***, donde n es el número de trabajos.

En cada iteración del bucle principal, se añade la nueva solución generada a la cola de prioridad, lo cual conlleva un costo de inserción en la cola de prioridad de O(log n), siendo n el número de soluciones almacenadas en la cola. Es por tanto que al estar anidados ambos bucles, acaban generando una complejidad para nuestro algoritmo de ***O(n log n)*** en el caso promedio.

Por otro lado, en el peor caso, todas las soluciones parciales tendrán que ser generadas y añadidas a la cola de prioridad. En este caso, el número máximo de soluciones parciales que pueden es igual al número total de permutaciones de los trabajos, que es **n!** . Sin embargo, el algoritmo esto lo conseguimos evitar gracias al criterio de poda que es lo que nos permite evitar expandir soluciones que ya tienen una penalización mayor que la mejor solución encontrada hasta el momento, lo cual con esto reducimos significativamente el número de soluciones que se generan y añaden a la cola de prioridad.

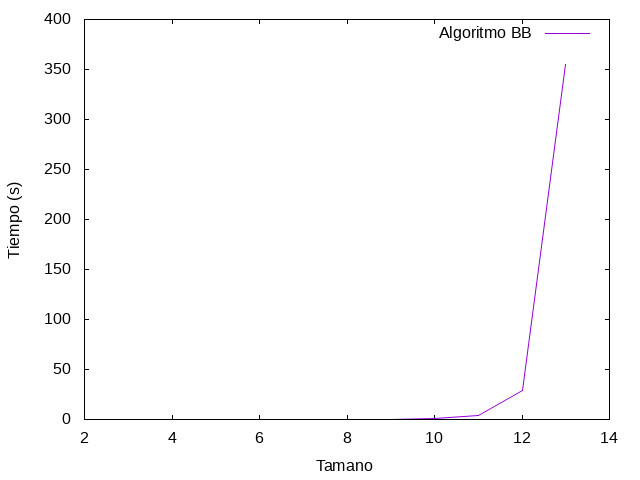
En resumen, el algoritmo utilizado ***algoritmoBB()*** tiene una complejidad promedia de ***O(n log n)*** y en el peor de los casos de ***O(n!)*** en términos del número de trabajos, teniendo en cuenta siempre que la eficiencia dependerá siempre de los datos y del estimador utilizado, ya que esto último afecta a la calidad de las podas realizadas.

## 1.3.1.2. Análisis empírico

Al igual que con el algoritmo Backtracking los resultados obtenidos se han conseguido a través de la media de 5 tiempos reales asociados a la ejecución del método. De esta forma conseguimos una mejor representación de la eficiencia del algoritmo.

| ***Nº de trabajos*** | ***Tiempos (s)*** |
| --- | --- |
| *3* | *2.8e-06* |
| *4* | *8.8e-06* |
| *5* | *3.7e-05* |
| *6* | *0.0002104* |
| *7* | *0.0014006* |
| *8* | *0.0091354* |
| *9* | *0.0731968* |
| *10* | *0.71873* |
| *11* | *4.28826* |
| *12* | *29.4199* |
| *13* | *355.152* |

Representación gráfica de los resultados obtenidos:

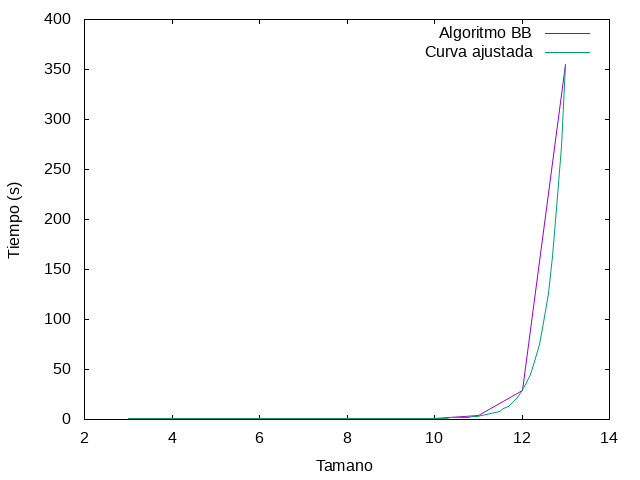


## 1.3.1.3. Análisis híbrido

Como hemos visto en el estudio teórico se trata de un algoritmo del modo n\*log(n), a pesar de esto consideraré otra función para el ajuste. Esto lo haré debido a que n\*log(n) es la eficiencia asociada a una rama, cuando luego realmente en la mayoría de casos tiene que recorrer más de una. Por lo que la función de ajuste será la siguiente:

A través del software *gnuplot*, deducimos por regresión mediante mínimos cuadrados que las constantes ocultas serán:

Representación gráfica:



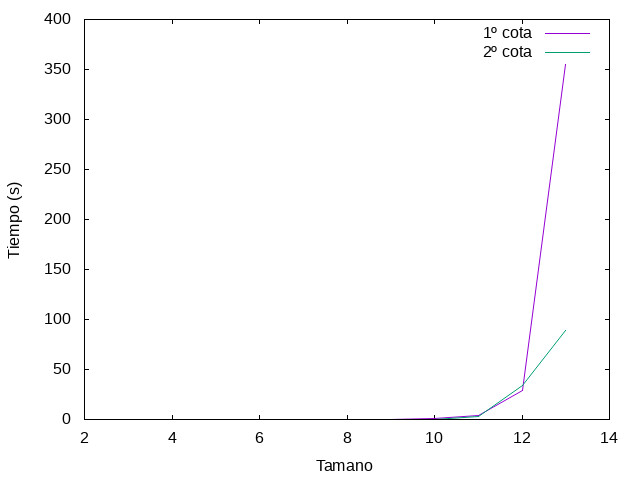
En este tipo de algoritmos, el tiempo de ejecución de cada conjunto de datos está determinado por diversos factores que impiden que se mantenga una tendencia. Esto se debe a que el número de operaciones vendrá determinado por el momento en el que cortemos en nuestro árbol, es por esto mismo que cada caso es totalmente particular lo que impide que se pueda comparar realmente con el resto de casos obtenidos.

# Estudio de cotas

Para finalizar el estudio de la técnica ***Branch and Bound*** usada en la implementación del algoritmo de nuestro ejercicio realizaremos una comparativa entre las dos cotas que hemos empleado:

1º. **Cota 0 (*estimador0()*)**→ Solo tiene en cuenta las penalizaciones acumuladas hasta el momento

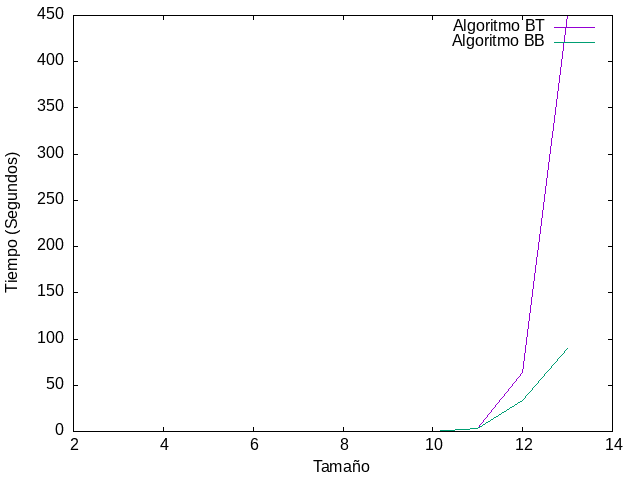
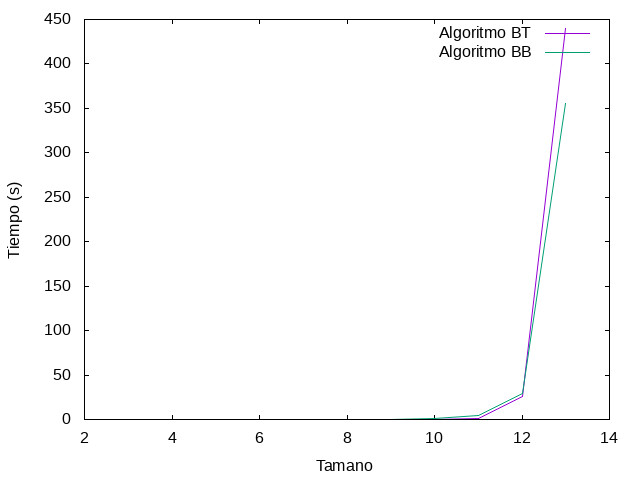
2º. **Cota 1 (*estimador1()*)** → Realiza una previsión en base a los plazos ya cumplidos.



Tras esta comparativa gráfica, vemos como la cota 2º resulta ser mucho mejor que la cota 1º ya que esta no realiza ningún tipo de estimación al contrario que la cota 2º que al sí realizarla permite una mejora de los cálculos y por tanto una mejora de la eficiencia del algoritmo.

* 1. Comparativa de algoritmos

Primero hemos realizado mediciones en ambos algoritmos utilizando *estimador0()* y más tarde otra con el segundo de ellos, obteniendo los siguientes resultados:



Como observamos, aunque con la primera gráfica no apreciamos gran diferencia, a la hora de usar el estimador 1, obtenemos unos tiempos bastantes mejores con el algoritmo que implementa ***Branch and Bound*** que con el *Backtracking*, sobretodo cuando se aplica sobre un gran número de trabajos.

En lo referente a las soluciones, no existe diferencia entre la eficacia de ambos, ya que llegan a una misma solución óptima.

1. Conclusiones

Implementando una solución para nuestro correspondiente problema utilizando ambas técnicas llegamos a la conclusión de que, aún a pesar de la complejidad de ambas soluciones, en especial de la que aplica *Branch and Bound*, estas resultan muy útiles en problemas que implican la exploración de árboles de estados.

Si no utilizáramos estas, los posibles algoritmos que se utilizaran para este proceso tendrían tiempos de ejecución demasiado grandes como para que fueran eficaces.

Por otro lado, cabe resaltar la importancia de desarrollar una buena forma de realizar una estimación en cada momento para establecer las cotas locales, ya que marca una gran diferencia en los tiempos de ejecución del algoritmo, debido a que se podarán más ramas y más grandes en función de cómo de buena sea la aproximación del peso de esta desde un principio.