DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

Algoritmos y estructuras de datos III

Integrante	LU	Correo electrónico
Valeria Arratia Guillen	467/21	arratiavale@gmail.com
Jennifer Potter	171/21	jennpjszk@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

1. Algoritmo Goloso

1.1. Descripción del Problema

El problema de selección de actividades consiste en encontrar dentro de un conjunto de actividades $A = \{A_1, ..., A_n\}$ un subconjunto de cardinalidad maxima tal que las actividades no se solapen en tiempo, teniendo en cuenta que una actividad puede comenzar en el instante que termina la anterior. Cada actividad A_i se realiza en un intervalo (s_i, t_i) siendo s su momento inicial y t su momento final. Por ejemplo, si tenemos 5 actividades que se realizan en los intervalos (6, 7), (1, 4), (0, 3), (4, 10) y (3, 6). El algoritmo debería elegir la mayor cantidad de actividades que no se solapen en el tiempo. Esto sería elegir los 3 intervalos (0,3), (3,6), (6,7), los cuales tienen los índices 3, 5 y 1.

1.2. Solución

La resolución planteada utiliza la tecnica algoritmica golosa. Primeramente, recibimos los intervalos de tiempo de las actividades y a cada una de estas les agregamos su índice y los guardamos en el vector. De esta manera, en el conjunto de actividades se guardan los tiempos de una actividad junto a su índice. Luego, la idea planteada fue ordenar todas las actividades según su momento de finalización en forma creciente utilizando el algoritmo de Bucket Sort. Esto con el objetivo de tener las actividades que terminan más temprano primero, para así, posteriormente, elegir y maximizar la cantidad de actividades que no se solapan en el conjunto solución.
En segundo lugar, elegimos la primera actividad del vector ya ordenado, la cual finaliza más temprano y, sin duda, es la primera mejor opción. Además, para guardar nuestra solución, utilizamos un contador para las actividades que vayamos seleccionando y un vector que guarda sus índices.
Una vez que elegimos la primera actividad, actualizamos el contador y guardamos su índice, recorremos el resto de las actividades comparando el momento inicial de la actividad a agregar con el momento final de la última actividad agregada. De tal manera que, si estas actividades no se solapan en tiempo, es decir, el momento inicial es mayor o igual que el momento final de la actividad anterior, entonces esa actividad es parte de la solución, en consecuencia, aumentamos el contador y guardamos su indice. En caso contrario, seguimos recorriendo el vector y repetimos este procedimiento hasta que llegamos al final de actividades.
Finalmente, una vez que terminamos imprimimos por pantalla el contador de actividades elegidas y el vector de sus índices.

1.3. Demostración del Algoritmo

Vamos a demostrar que nuestro algoritmo es correcto. Para esto, usamos inducción en cada paso i. Veamos que la solución parcial $B_1, ..., B_i$ que brinda el algoritmo goloso en el paso i, se puede extender a una solución óptima $S = B_1, ..., B_i, C_{i+1}, ..., C_j$

Caso Base:
$P(0)$: Cumple. Dado que la solución greedy se encuentra vacia, es evidente que es extensible a una solución optima cualquiera

Paso Inductivo:
$P(i) \Rightarrow P(i+1)$
Por hipotesis inductiva, como $i > 0$ tenemos que el conjunto $B = B_1, ..., B_i$ que brinda el algoritmo goloso en el paso i se puede extender a una solución óptima $S = B_1, ..., B_i, C_{i+1}, ..., C_j$. Por lo tanto, existe un conjunto $C = C_{i+1}, ..., C_j$ con al menos un elemento tal que $B \cup C = S$.

Por consiguiente, queremos probar que la solución parcial $B = B_1, ..., B_i, B_{i+1}$ se puede a una solución óptima $S = B_1, ..., B_{i+1}, C_{i+2}, ..., C_j$

Dada la hipotesis inductiva, sabemos que existe un conjunto C con al menos un elemento que extiende al conjunto B a una solución óptima. Queremos ver que en el paso $i+1$ esto sigue siendo posible. Pueden pasar 3 cosas:

Caso 1:
La actividad B_{i+1} que queremos agregar se superpone en tiempo con la anterior agregada B_i, es decir $B_{i+1}.tiempoInicial < B_i.tiempoFinal$. Por lo tanto, B_{i+1} no pertenece a la solución óptima, pues si sí perteneciera, habría que sacar, por lo menos, el último agregado B_i y no sería una mejor solución, dado que buscamos maximizar la cantidad de actividades en la solución. De este modo, en el paso $i+1$, llegamos con la secuencia de actividades $B = B_1, ..., B_i$ que por hipotesis inductiva es extensible a una solución óptima $S = B_1, ..., B_i, C_{i+1}, ..., C_j$

Caso 2:
La actividad B_{i+1} que queremos agregar no se superpone en tiempo con la anterior agregada B_i, es decir, $B_{i+1}.tiempoInicial \geq B_i.tiempoFinal$. Por hipotesis inductiva, sabemos que $B = B_1, ..., B_i$ que brinda el algoritmo goloso en el paso i se puede extender a una solución óptima $S = B_1, ..., B_i, C_{i+1}, ..., C_j$. Puede pasar que B_{i+1} se superpone en tiempo con C_{i+1}, en ese caso la secuencia de actividades con la que llegamos a $i+1$ es $B = B_1, ..., B_i, B_{i+1}$. Y como C_{i+1} es B_{i+1}, entonces la extensión $B_1, ..., B_i, B_{i+1}, C_{i+2}, ..., C_j$ es igual a la solución S y, por lo tanto, es valida y óptima.

Caso 3:
Para el caso en el que la actividad B_{i+1} no se superpone en tiempo con la anterior agregada B_i. Puede pasar que B_{i+1} no se superponga en tiempo con C_{i+1} pero eso sería un absurdo pues no sería la solución supuesta por la hipotesis inductiva.

1.4. Complejidad

Para determinar la complejidad computacional de nuestro algoritmo tenemos que analizar las distintas operaciones que se realizan en este.

Primeramente, la utilización del algoritmo de Bucket Sort para el ordenamiendo del vector de actividades, tiene una complejidad de $O(N)$, con N igual al número de actividades a ordenar según su tiempo de terminación. Conviene aclarar que Bucket Sort tiene una complejidad de $O(N + K)$ con N igual al tamaño del vector a ordenar y K igual a la cantidad de buckets. En esta implementación, ya que conocemos la cota $2 * N$ del tiempo maximo que puede tomar cada actividad, se utilizaron 2*N buckets para ordenar el vector. Es decir que el ordenamiendo del vector de actividades tiene una complejidad $O(N + 2 * N) = O(3 * N) = 3 * O(N) \subseteq O(N)$

Posteriormente, el recorrido del vector se realiza en O(N), y el algoritmo elige una actividad o no según su solapamiento en O(1). Con lo cual, esto nos lleva a una complejidad lineal O(N).

En definitiva, el algoritmo planteado presenta una complejidad temporal lineal $O(N)$.

1.5. Experimentación

Realizamos una experimentación con un conjunto de instancias para mostrar que el algoritmo tiene la complejidad mencionada anteriormente.

Usamos distintos tamaños de la entrada y calculamos los distintos tiempos de ejecución que tarda nuestro algoritmo. Al realizar esto, podemos ver en este gráfico cómo va aumentando el tiempo de ejecución a medida que vamos incrementando la cantidad de actividades en la entrada del algoritmo.

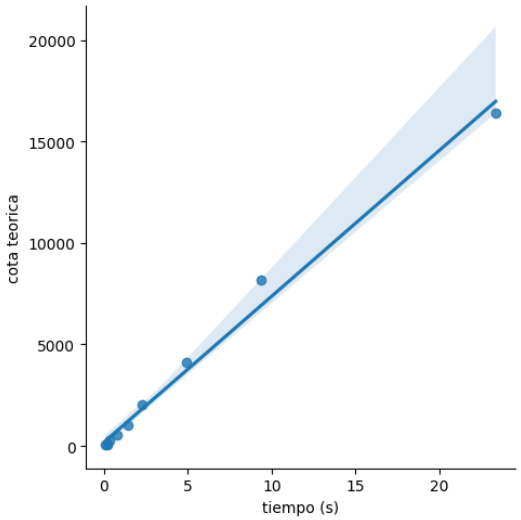

Figura 1

Este gráfico tiene el problema de que los n que consideramos (el tamaño de la entrada) crecen exponencialmente. Podemos hacer el gráfico con escala logaritmica para que se vea mejor.

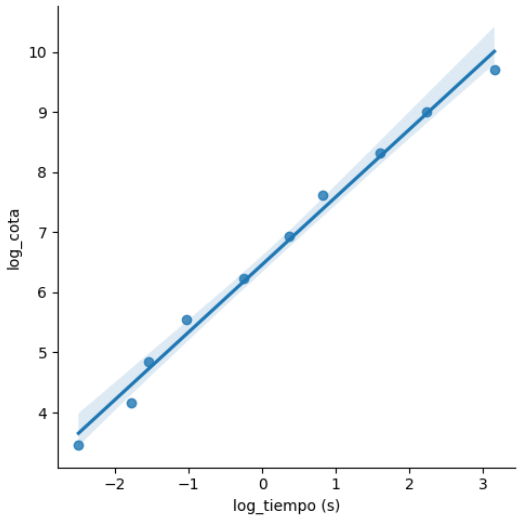

Figura 2

En este gráfico se puede apreciar mejor los tiempos de ejecución del algoritmo. Además, el coeficiente de correlación de Pearson es 0.9954960902447488, este nos indica que mientras este más cerca del 1, tiene mayor dependencia lineal entre variables.

Podemos concluir que hay un alto nivel de confianza que la cota teórica correlaciona con los tiempos de ejecución que vimos.

Por otro lado, realizamos la experimentación comparando los casos en los que la entrada ya viene ordenada y en los que no. Tomamos los valores con escala logarítmica para que se aprecie mejor

esta diferencia.

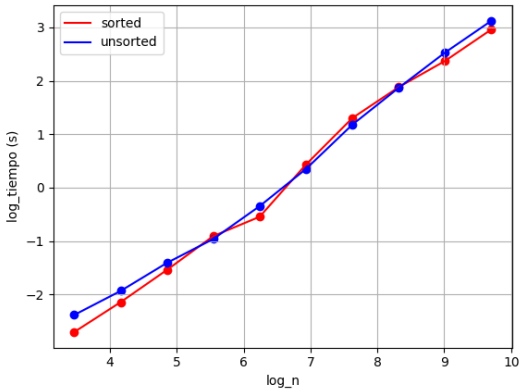

Figura 3

En el gráfico podemos observar que la diferencia de tiempo entre los dos casos que propusimos no es tanta, incluso en algunos casos se puede ver que las instancias ordenadas tardan más que las desordenadas.

1.6. Preguntas

Existe un conjunto de instancias que sean más fáciles de resolver? Existe un conjunto de instancias que sean más difíciles de resolver? En otras palabras, si fijamos el tamaño de instancia, podemos construir los si y ti de tal forma que podamos decir algo del comportamiento del algoritmo?

Si fijamos el tamaño de la instancia y construimos los s_i y t_i de manera tal que esten ordenados por tiempo de finalización y ninguno se solape en tiempo podemos concluir que el algoritmo siempre va a agregar al resultado los índices de todas las actividades en el orden ingresado al inicio de la ejecución. Y el caso en el que la entrada es ingresada de forma tal que los s_i y t_i esten ordenados decrecientemente por tiempo de finalización y ninguno se solape, podemos concluir que el algoritmo siempre va a agregar al resultado los índices de todas las actividades en orden decreciente al ingresado al inicio de la ejecución.