DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico I

Técnicas Algorítmicas

23 de abril de 2023 — Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Teplizky, Gonzalo Hernán	201/20	gonza.tepl@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

Resumen

Para este trabajo práctico el enfoque será puesto en utilizar distintas técnicas algoritmicas vistas en la materia, con el objetivo de encontrar algoritmos más rápidos y eficientes que aquellos que emplean fuerza bruta. Mas allá de que en la mayoría de los casos es imposible evitar una complejidad exponencial, se intentará acotar lo más posible el tiempo de cómputo de cada programa implementado en este trabajo. Se deben resolver tres ejercicios donde, en cada uno de ellos, se analizará el comportamiento de cada una de estas tres técnicas en específico: backtracking, programación dinámica y, por último, goloso/greedy.

En cuanto a los ejercicios de este trabajo, el desempeño de las implementaciones será testeada por un *juez online*[1].

Este informe se centrará en el ejercicio propuesto para la tercer técnica algorítmica mencionada, la golosa. La estructura es la siguiente:

- **Introducción**: Se da un vistazo inicial del problema a resolver.
- **Desarrollo**: Se explica el algoritmo desarrollado para la resolución del problema, justificando la complejidad computacional y la correctitud del mismo.
- **Experimentación y resultados**: Se justifica empíricamente porque la complejidad es la provista, considerando distintas instancias y distintos tamaños.

Palabras clave: *Fuerza Bruta*, *Backtracking*, *Greedy*, *Programación Dinámica*, *Complejidad Temporal*, *Complejidad Espacial*.

Índice

[1] `https://www.spoj.com/tutorials/`

1. Ejercicio 3

1.1. Presentación del Problema

En este ejercicio se nos presenta un conjunto de actividades A. Cada actividad a_i se representa mediante un intervalo $[s_i, t_i]$, siendo s_i y t_i valores discretos enteros positivos, que representan el inicio y el final de cada actividad.
Con respecto a la duración de los intervalos, sabemos que vale que $0 \leq s_i < t_i \leq 2N$, siendo N el cardinal del conjunto inicial de actividades A.
A partir de estas actividades debemos hallar el subconjunto de máxima cardinalidad C a partir de A que este conformado únicamente por actividades cuyos tiempos no se solapen.
En este caso particular, se establece que una actividad puede comenzar en el preciso instante en que termina la anterior.

1.2. Algoritmo

En este caso, procederemos a resolver el ejercicio bajo la técnica Golosa o Greedy, donde construimos un procedimiento heurístico que pueda llevarnos a un algoritmo que encuentre alguna solución optima para el problema.
A diferencia de otras técnicas algorítmicas como Programación Dinámica, el enfoque aquí es el de encontrar alguna estrategia o decisión "golosa" la cual tomaremos en cada paso hasta alcanzar una solución. Son decisiones locales, que primero tomamos, y luego pasamos a resolver el subproblema restante.
Siguiendo el enunciado de la práctica, consideraremos la siguiente estrategia golosa: de entre todos los intervalos que todavía no fueron considerados, proponemos agarrar aquel cuya finalización sea lo más temprano posible, pero sin solaparse, es decir, que empiece (al menos) en el instante que finaliza la última actividad que seleccionamos para el subconjunto C.
Para que esta idea cobre sentido, observemos que los intervalos de A deben estar ordenados crecientemente por el t_i de cada actividad a la que representan (su instante final). Esta será nuestra hipótesis.
Supongamos entonces que vamos recorriendo las actividades de A. Fijamos la primera actividad de este conjunto (ya ordenado por los intervalos derechos de sus elementos). Teniendo una primera actividad fijada a C, avanzamos hacia la siguiente actividad de A. Por hipótesis no puede suceder que la actividad actual termine antes que la anterior. Luego, podemos notar que se cumplen cualquiera de los siguientes escenarios:

- La actividad actual inicia **en el momento en que termina la anterior**, o unos instantes después. Por hipótesis podemos afirmar que no existe una actividad que termine antes que la actual, a lo sumo termina en el mismo momento. Por lo tanto, podemos agregar a la actividad actual al conjunto C, cumpliendo así con la estrategia golosa.
- La actividad actual inicia **antes que la actividad previa termine**. Esto implica que esta actividad se solapa con la anterior. Por hipótesis sabemos que la actividad actual termina luego de la anterior, o en el mismo momento que esta. Pero en ningún escenario va a ser más óptima que la anterior, y no podemos garantizar que sea igual. Es por esto que la descartamos y pasamos a la siguiente actividad de A.

Un posible pseudocódigo para representar esta idea, es el siguiente:

Algorithm 1 seleccionarActividades(A):

```
C ← S ∪ A_0
actividadAnterior ← A_0
i ← 1
while i < |A| do
  if noSeSolapan(ActividadAnterior, A_i)
    C ∪ A_i
    actividadAnterior ← A_i
  i ← i + 1
end while
return C
```

Para que este algoritmo cobre sentido, debe valer la pre-condición del ordenamiento ascendente de las actividades de A por su intervalo derecho.

1.3. Correctitud del Algoritmo

Queremos demostrar que el problema de elegir un subconjunto de actividades C a partir de un conjunto de actividades A, que sea maximal y cuyos tiempos no se solapen, puede resolverse y sea correcto con la estrategia golosa propuesta.
Para eso, queremos demostrar sobre la solución:

(1) Que cumplimos la propiedad de la decisión golosa: toda solución óptima S = $s_1, s_2, ..., s_m$ de actividades de máxima cardinalidad que no se solapan, puede modificarse para algún intervalo k, pasando a usar en ese punto k una elección golosa, y la solución sigue siendo óptima.

(2) Que posee una subestructura óptima: una solución óptima para el problema puede obtenerse concatenando las soluciones óptimas de los subproblemas que van quedando luego de cada elección.
En este caso, queremos ver que la secuencia de k decisiones golosas $G_k = g_1, g_2, .., g_k$ puede ser extendida a una solución óptima, "concatenándose" a la óptima del subproblema $S_{[k+1,m]}$, el subconjunto máximal de actividades no solapadas que empieza, al menos, desde que termina la actividad k y llega hasta que termine la última actividad, todo bajo el orden creciente por finalización de actividad, con el que vamos alcanzando cada intervalo, propuesto en la etapa algorítmica.

Comencemos **probando (1)** de forma directa.
Para esto, tomamos alguna solución óptima $S = s_1, s_2, ..., s_k, s_{k+1}, ..., s_m$ y la modificamos utilizando g_k, la k-ésima decisión golosa, quedandonos $S' = s_1, s_2, ..., g_k, s_{k+1}, ..., s_m$. En particular, por decisión golosa, sabemos que $g_k = min_j\{\ t_{k-1} \leq s_j\}$, la actividad que antes comienza entre todas las que no se solapan a la k-1, la última seleccionada.
S' sigue siendo óptima ya que sabemos que $t_{g_k} \leq t_{s_k}$ y como ya valía que $t_{s_k} \leq s_{s_{k+1}}$ ya que S es óptima y ninguna actividad se solapa, entonces también vale $t_{g_k} \leq s_{s_{k+1}}$. De esa forma, utilizando la elección golosa y reemplazándola en la solución óptima, no solapamos intervalos y mantenemos un conjunto de igual cardinal. Así, dejamos probado que podemos modificar cualquier solución óptima modificando alguna actividad por una extraída de la secuencia de decisiones golosas, y seguimos en una solución óptima. □

Para **probar (2)**, haremos inducción en **k**.

Sea la proposición P(k) ≡ G_k es la secuencia de k decisiones golosas la cual puede ser extendida a una solución óptima.

- **Caso Base:** P_0 es trivial porque $B_0 \equiv \emptyset$, es decir, no tome aún ninguna decisión. El

subproblema restante es realmente el problema completo. Si la óptima existe, podré extender desde el $\emptyset$ a ella.

- **Paso inductivo:** Queremos ver que $P(k)$ implica $P(k+1)$.
  Si tomamos a m como el cardinal de la solución óptima, entonces vamos a asumir que queremos probar el paso inductivo para cuando $k+1 \leq m$, ya que en caso contrario, se cumple directamente, debido a que G_k ya sería una solución óptima de cardinal máximo. Entonces, queremos probar que si dada una secuencia de k decisiones golosas extensibles a una solución óptima, con $k+1$ decisiones también podemos lograrlo.

  Sabemos por **HI** que $g_k = g_1, g_2, ..., g_k$ se puede extender a una solución óptima $S = g_1, g_2, ..., g_k, s_{k+1}, s_{k+2}, ..., s_m$. En particular, esto nos dice que vale $t_{g_k} \leq s_{s_{k+1}}$ y $t_{s_{k+1}} \leq s_{s_{k+2}}$. Ninguna de las 3 actividades, g_k, s_{k+1} y s_{k+2} se solapan. Tomamos $g_{k+1} = min_j\{\, t_k \leq s_j\}$, ésta es la primer actividad en no solaparse con la última seleccionada, la k, que es posterior a ella.
  Sabiendo que $t_{g_k} \leq t_{g_{k+1}}$ y que $t_{g_{k+1}} \leq s_{s_{k+1}}$, entonces se cumple que $t_{g_{k+1}} \leq s_{s_{k+2}}$, llegando a la solución óptima $S' = g_1, g_2, ..., g_k, g_{k+1}, s_{k+2}, ..., s_m$, de máxima cardinalidad y sin actividades solapadas, donde la g_{k+1} ya de por sí estaba acotada por la s_{k+1} original. Así probamos que G_{k+1} se extiende a una solución óptima, gracias a la hipótesis previa y a la posibilidad de reemplazar en cualquier paso por una decisión golosa, probado en **(1)**. □

1.4. Complejidades del Algoritmo

Para que valga la hipótesis debemos ordenar previamente los intervalos de A por su componente de cierre. Descartamos los algoritmos de ordenamiento por comparación, dado que estos tienen como cota de complejidad de peor caso inferior O(N log N), siendo N la cantidad de actividades del conjunto inicial A.
Observamos que vale que $0 \leq s_i < t_i \leq 2N$, es decir, sabemos de antemano que los comienzos de las actividades, a lo sumo, serán dos veces la cantidad de actividades que haya. Encontramos así **una cota** para los intervalos, ya que su valor depende linealmente del tamaño de la entrada.
Es por esto que podemos recurrir al algoritmo de Bucket Sort, que nos servirá para poder ordenar los elementos en $O(2N) = O(N)$ y así alcanzar la complejidad temporal lineal esperada.
Luego, notemos que seleccionarActividades(A) recorre una única vez cada actividad del conjunto A. El procesamiento de cada actividad requiere chequear ciertas condiciones de comparación entre enteros positivos, las cuales asumimos de tiempo de peor caso constante. Por lo tanto, esta complejidad se sigue manteniendo a lo largo de todo el procesamiento. La complejidad final del algoritmo es de $O(N)$

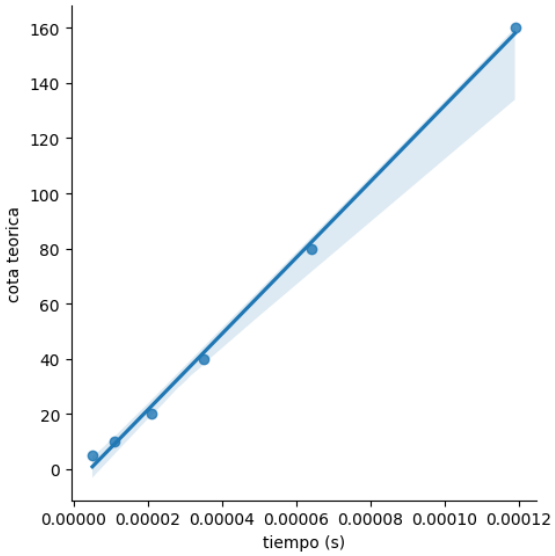
1.5. Experimentación

Para finalizar, realizamos una pequeña experimentación donde a través de instancias específicas, es decir, conjuntos de actividades de distintos tamaños y diversos intervalos, buscamos **ratificar empíricamente** la complejidad de $O(N)$ del algoritmo.

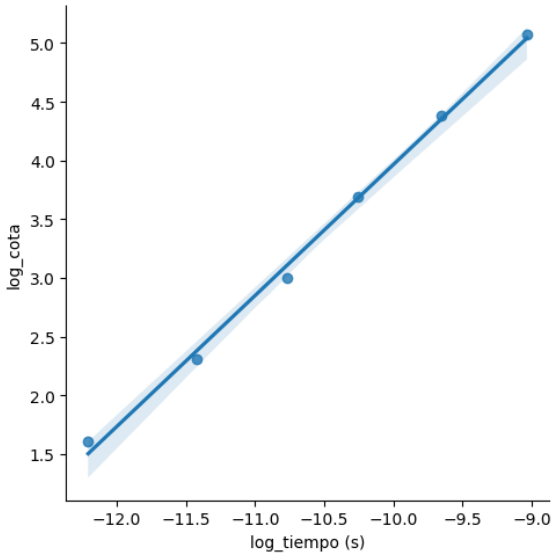
Para ver entonces que el algoritmo crece linealmente en su tiempo de ejecución por el tamaño de la entrada, armamos algunas instancias de tamaños que aumentan por 2.
Elegimos en particular ir moviendo al n desde n=5, a n=10, n=20, n=40, n=80 y hasta n=160.
Utilizando la experimentación provista en la clase práctica de Greedies, armamos dos gráficos para conocer la relación entre el tiempo de ejecución y el tamaño de la entrada, comprobando efectivamente el crecimiento lineal en la ejecución.

En el primero de ellos, construimos un gráfico de dispersión con una regresión lineal y un ajuste

de cota teórica, según los tamaños de entrada elegidos. La regresión devuelve la recta formada por el eje x que es el tiempo, y el eje y que es el tamaño. La pendiente de la línea representa la tasa de crecimiento del tiempo en función del tamaño de la entrada.



El segundo gráfico representa la relación lineal entre el logaritmo del tiempo de ejecución y el logaritmo de la cota teórica. Ayuda a mejorar la precisión y ver más claramente lo que habíamos afirmado en la teoría: que **el algoritmo tiene una complejidad lineal**.



Por último, en cuanto a medir el tiempo de ejecución a nivel de las instancias en sí y no del tamaño de las mismas, **no sucede aquí** un fenómeno como en el de meximización, donde el algoritmo se basa en ir generando una escalera, un ordenamiento, y claramente resulta más rapido cuando está ordenado o prácticamente ordenado de antemano.
En este caso, el algoritmo para hallar el máximo conjunto de actividades que no se solapan entre sí,

realiza siempre el ordenamiento previo por el tiempo final de cada actividad utilizando Bucket Sort, sin chequear por ejemplo, que ya esté ordenado (lo cual también tendría costo lineal) de manera que no resulta de más ayuda que venga ordenado desde el input, ni tampoco nos cambiarían situaciones como tener todos intervalos solapados de modo que solo me quedo con el primero, por dar un ejemplo. A lo sumo evitamos operaciones que son constantes como la de agregar un elemento más a la estructura de la solución.