DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

TP 1: Técnicas Algorítmicas

23 de Abril, 2023 — Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Zaid, Pablo	869/21	pablozaid2002@gmail.com
Arienti, Federico	316/21	fa.arienti@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

Una estrategia *golosa* —o *greedy*— es una estrategia de resolución de problemas, por lo general de optimización, que aprovecha las propiedades intrínsecas a una solución óptima para lograr resolver un problema de manera correcta y computacionalmente eficiente. En relación a la estrategia de *programación dinámica*, el método se basa[1] en la aplicación de la propiedad de *subestructura óptima*[2]: una solución óptima a un problema incorpora soluciones óptimas a subproblemas relacionados. Sin embargo, a diferencia de esta estrategia, y del método más general de *backtracking*, los algoritmos *greedy* obtienen una solución por medio de decisiones que "*contemplan la información inmediatamente disponible, sin preocuparse por los efectos que estas decisiones puedan tener en el futuro*"[3]. Un comportamiento que requiere que el problema tenga la propiedad de *selección golosa*: una solución globalmente óptima se puede obtener a partir de decisiones localmente óptimas.

El siguiente informe evalúa la aplicación del método sobre el problema de la *selección de actividades*, que se enmarca dentro del más general *problema de la fiesta*. Además, evalúa la eficiencia del algoritmo resultante de manera empírica.

Palabras clave: *Algoritmos golosos, estrategias algorítmicas, selección de actividades.*

CONTENIDOS

[1]Ver Thomas H. Cormen; Charles E. Leiserson; Ronald L. Rivest y Clifford Stein. Introduction to algorithms. 2009. Sección 16.2: *Elements of a greedy strategy.*

[2]No todo problema tiene este propiedad y no todo problema que la posee tiene una solución *greedy.*

[3]Ver Gilles Brassard y Paul Bratley. Fundamentals of Algorithmics. 1995. Capítulo 6: *Greedy Algorithms.*

1. El problema de la selección de actividades

El problema de la selección de actividades que consideraremos tiene la siguiente premisa. Dado un conjunto

$$A := \{a_1 \ ... \ a_n\}$$

de actividades disponibles, donde, para todo $1 \leq i \leq n$, la actividad a_i se representa por el intervalo $[s_i,\ t_i)$ —que corresponde, respectivamente, al tiempo de inicio y finalización de la actividad—, queremos encontrar un subconjunto de A que nos permita realizar la mayor cantidad de actividades posibles.

En esto, queremos elegir las actividades con la restricción que, para todo par de actividades a_i y a_j seleccionadas, $1 \leq i,\ j \leq n$, a_i y a_j no se solapen en el tiempo. Es decir, $s_i \geq t_j$ o $s_j \geq t_i$. Por ejemplo, si

$$A := \{[1,\ 3),\ [1,\ 4),\ [3,\ 6),\ [4,\ 7),\ [7,\ 8)\}$$

luego las soluciones óptimas son $\{[1,\ 3),\ [3,\ 6),\ [7,\ 8)\}$ y $\{[1,\ 4),\ [4,\ 7),\ [7,\ 8)\}$.

Como condición extra, limitaremos los tiempos para las actividades de manera tal que $1 \leq s_i < t_i \leq 2n$.

1.1. Demostración de la propiedad de subestructura óptima.

Como preámbulo a la aplicación de una estrategia *golosa* para resolver el problema, veamos primero que el problema tiene la propiedad de *subestructura óptima*.

Demostración. Consideremos una solución óptima $S \subset A$ que contiene, sin pérdida de generalidad, a alguna actividad $a_i \in A$ para $1 \leq i \leq n$. Luego, podemos particionar tanto a S como a A entre aquellas actividades que terminan antes que comience a_i y aquellas que comienzan después de que termine a_i.

Si estas particiones de S no son, correspondientemente, óptimas para las particiones de A, entonces existe algún subconjunto de las actividades que terminan antes que comience a_i que tiene un tamaño mayor que la partición de S que consideramos y, de igual manera, existe una solución mayor para el subconjunto de actividades que comienzan después de que termine a_i.

Sigue entonces que S no es óptima, ya que podemos formar una solución mejor a partir de estos otros dos subconjuntos, lo que es un absurdo. □

1.2. Demostración de la propiedad de selección golosa.

Consideremos ahora la siguiente estrategia *golosa*: elegir la actividad que finalice primero, de entre todas las actividades que sigan disponibles. Vamos a demostrar que esta estrategia es correcta.

Demostración. Notemos que, si la solución parcial $B_1 \ ... \ B_i$ que brinda el algoritmo goloso en el paso i, para todo $0 \leq i \leq n$, se puede extender a una solución óptima, entonces la estrategia es correcta. Lo demostraremos por inducción.

Para el caso base, $i = 0$, el algoritmo todavía no eligió ninguna actividad. Luego, podemos extender la solución a una solución optima de manera trivial.

Supongamos ahora, para $i > 0$, que tenemos una solución parcial $B_1 \ ... \ B_i$ de actividades elegidas por nuestro algoritmo que se puede extender a una solución óptima

$$B_1 \ ... \ B_i,\ C_{i+1} \ ... \ C_j$$

donde, sin pérdida de generalidad, vamos a suponer que la secuencia de actividades sin solapamiento C_{i+1} ... C_j está ordenada por tiempo de finalización.

Notar que, por nuestro método de selección, B_1 ... B_i también debe estar ordenado de la misma manera. Luego, todas las actividades C_{i+1} ... C_j deben empezar después de que termine B_i para que la extensión sea válida. Si no, habría solapamientos, ya que, por definición de la estrategia, deben terminar después que B_i.

Consideremos ahora la solución parcial golosa B_1 ... B_{i+1}, donde B_{i+1} es la actividad cuyo momento final ocurre antes entre todas las actividades restantes y no se solapa con las actividades ya seleccionadas.

Como B_{i+1} debe terminar antes que C_{i+1} o $B_{i+1} = C_{i+1}$, entonces B_{i+1} no puede solaparse con ninguna actividad C_{i+2} ... C_j. Esto se debe a que C_{i+1} no lo hacía y, por hipótesis inductiva, es la actividad que termina antes en la extensión. Luego, B_1 ... B_{i+1} se puede extender por reemplazo directo a la solución óptima

$$B_1 \text{ ... } B_{i+1},\ C_{i+2} \text{ ... } C_j$$

□

1.3. **El algoritmo.** En base a estas propiedades, podemos considerar el siguiente algoritmo.

```
1 proc act(A: ℕ × ℕ) → ℕ × ℕ:
2     ordenar A por tiempo de finalizacion
3     P ← ∅
4     i ← 0
5     para (s, t) en A:
6         si s ≥ i:
7             P ← P ∪ {(s, t)}
8             i ← t
9     retornar P
```

Algoritmo 1. Pseudocódigo para la *Selección de actividades*.

El mismo itera sobre las actividades de A en orden de finalización. En base a la estrategia golosa, decide incluir, o no, una actividad si y sólo si comienza después de que termine la última actividad que ya incluyó, cuyo tiempo de finalización se guarda en la variable i. Dado que no tiene sentido considerar tiempos negativos, inicializamos i en 0.

1.4. **Complejidad temporal y espacial.** El comportamiento de *act* depende del método de ordenamiento que elijamos y de las características de la estructura subyacente al conjunto P (en particular, si la inserción de un elemento se puede realizar en tiempo constante).

Respecto al método de ordenamiento, dado que los tiempos de finalización estan acotados por $2n$, podemos utilizar *counting sort* para garantizar una complejidad de peor caso, tanto temporal como espacial, de orden lineal.

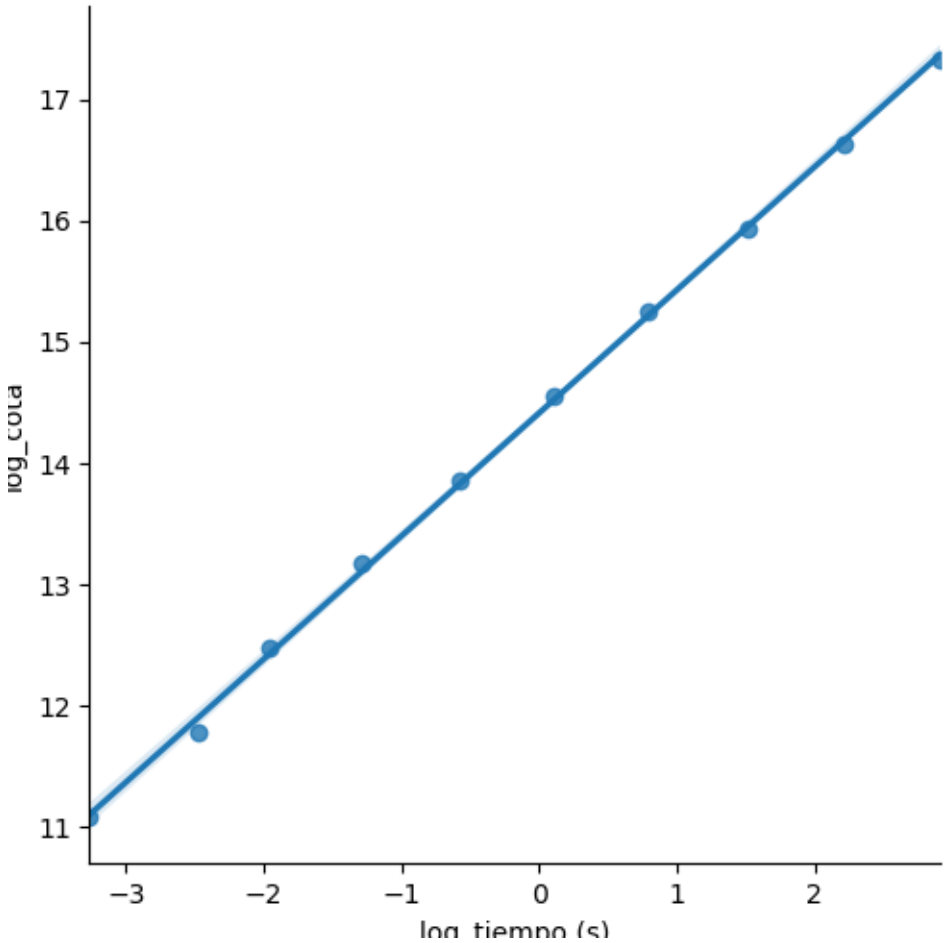
Respecto a las características de P, si se implementa como un arreglo de tamaño n —dado que una solución óptima tiene a lo sumo tamaño n—, sigue que el costo de inserción será constante. Sin embargo, deberémos tener cuidado de liberar el espacio excedente antes de retornar.

De estas observaciones, y el hecho de que, como son n actividades, el costo temporal del ciclo es $O(n)$, sigue que tanto la complejidad temporal como espacial de peor caso de *act* es $O(n)$.

2. Evaluación empírica

Para revisar empíricamente la cota lineal del algoritmo estipulado, primero tomamos los tiempos de ejecución de muestras aleatorias de tamaños de potencias de 2, desde la decimosexta hasta la vigésimosexta.

Luego graficamos el tiempo de ejecución de en función de estos tamaños de entrada, aplicando el logaritmo en ambas entradas para mejorar la visibilidad, dada la naturaleza exponencial de las muestras. Tras aplicar regresión lineal, el gráfico quedó de la siguiente forma:

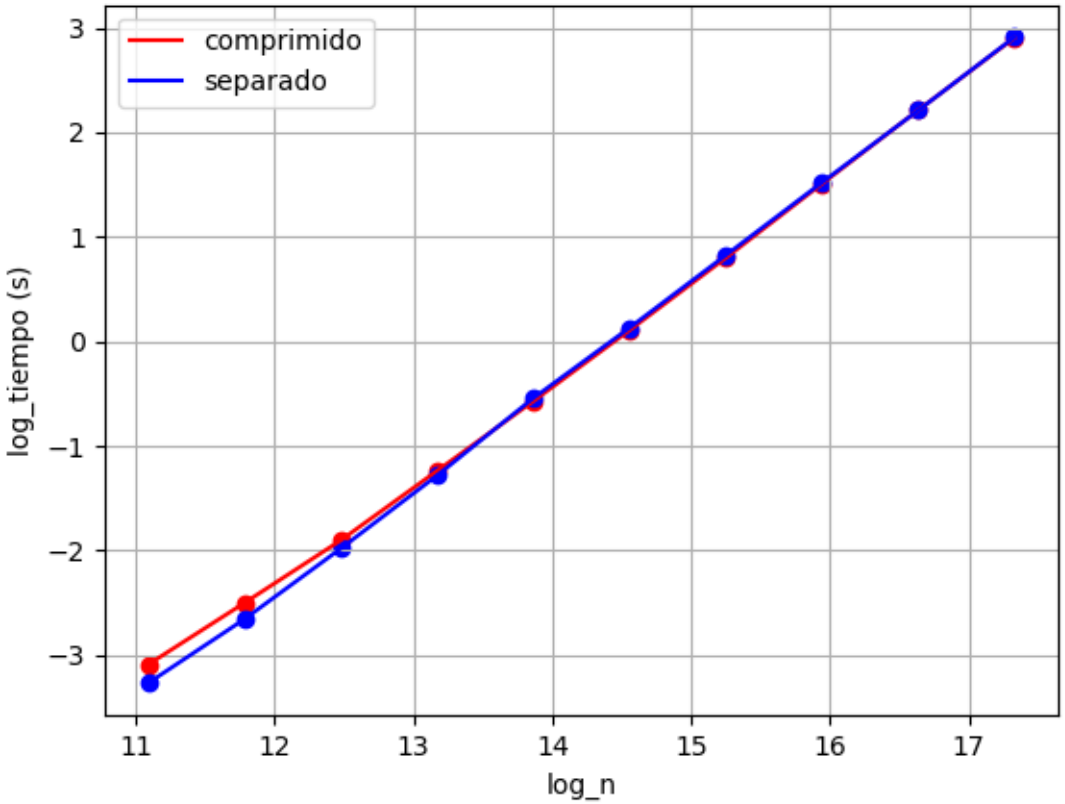


La relación lineal es bastante clara, pero calculamos también el coeficiente de Pearson para asegurarnos. Queda 0.9996692305026845, muy cercano a 1.

2.1. **Instancias de interés.**

Lo único que parece alterar el flujo del código en función de la entrada es si cada tupla ordenada empieza después (o al mismo tiempo) que cuando termina la última tarea de P. En ese caso se ejecutan dos lineas más, aunque tienen tiempo de ejecución constante y no parecen ser demasiado significativas.

Para probar la diferencia de tiempos entre instancias, hacemos el mismo test de la sección anterior, primero con una instancia comprimida en la que la intersección entre todos los intervalos es no vacía (sólo se entra al if en la primera ejecución del ciclo) y luego una instancia de entradas separadas donde la i-ésima entrada tiene la forma (i,i+1), por lo que siempre se entra al if. El gráfico queda de la siguiente forma:



Si bien es verdad que en las entradas más grandes el sample de entradas separdas parece haber tomado un poco más de tiempo, no resulta una diferencia significativa.