Algoritmos y Estructuras de Datos III

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Informe sobre el algoritmo goloso utilizado para el problema de la selección de actividades

Integrante	LU	Correo electrónico
Fabian Gómez Laguna	522/20	fabiangl2011@gmail.com
Nicolás Sergio Aquino	290/20	aquino_nicolas@hotmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

1. Presentación y descripción del problema

El problema sobre el que se proyectará este informe (conocido como el *problema de selección de actividades*) busca obtener a partir de un conjunto de actividades $\mathcal{A} = \{A_1, \ldots, A_n\}$, en donde cada actividad tiene horario de inicio s_i y final t_i con $0 \leq s_i < t_i \leq 2n$, un subconjunto $\mathcal{S}$ del mismo que cumpla las siguientes condiciones:

§ $\mathcal{S}$ es de cardinalidad máxima

Es decir, no existe otra solución válida con más actividades que $\mathcal{S}$.

§ No hay superposición de actividades.

Una actividad puede empezar en el instante que termina la anterior.

Uno podría pensar a priori en utilizar algunos de los algoritmos vistos en la materia, como por ejemplo *backtracking*, para resolver el problema. Si bien es una forma válida de resolver el problema, un algoritmo de *backtracking* resultaría en una complejidad de $\mathcal{O}(2^n)$, que es bastante ineficiente. Vamos a presentar y explicar un algoritmo *goloso* (o *greedy* en inglés) que resuelve este problema rápidamente y también vamos a demostrar que de hecho el algoritmo devuelve siempre una solución óptima.

2. Explicación del algoritmo

Sea $\mathcal{A} = \{A_1, \ldots, A_n\}$, donde $A_i = (s_i, t_i)$, $0 \leq s_i < t_i \leq 2n$ $\forall 1 \leq i \leq n$. Buscamos un $\mathcal{S} \subseteq \mathcal{A}$ de cardinalidad máxima tal que no se superpongan actividades. Dos actividades A_i, A_j, $i \neq j$, entraran en conflicto cuando $s_j < t_i$.

Observemos que dado un subconjunto cualquiera $\mathcal{A}' \subseteq \mathcal{A}$ podemos ordenarlo ascendentemente en función de t_i, es decir en función de sus tiempos de finalización y de esta forma podemos verificar que no haya superposición simplemente viendo que $t_i \leq s_j$, $\forall 1 \leq i < j \leq \#\mathcal{A}'$. Observemos ahora que para cualquier solución candidata $\mathcal{S}'$ del conjunto $\mathcal{A}'$, si asumimos que $\mathcal{S}'$ también está ordenado bajo el mismo criterio, se cumple que $t_i \leq s_{i+1}$ $\forall 1 \leq i < \#S'$.

Por lo tanto, para encontrar la solución que nos piden (la óptima) basta con elegir alguno de los conjuntos $\mathcal{S}'$ que tienen cardinalidad máxima. En otras palabras, podemos primero ordenar de forma ascendente el conjunto $\mathcal{A}$ e ir comparando el tiempo de inicio de la actividad actual con el tiempo final de la última actividad agregada a S y agregarla si no se solapa con esa última actividad. Esto se puede ver mejor con el siguiente pseudocódigo[1]:

```
ordenar(A)
B ← {A_1}
ultimo ← t_1
for 2 ≤ i ≤ n do
  if s_i ≥ ultimo
  then
    ultimo ← t_i
    B ⨁ A_i
  end if
end for
return B
```

3. Demostración del algoritmo

Supongamos entonces que $\mathcal{A}$ está ordenado en función de sus t_i con $1 \leq i \leq n$. y que cualquier solución candidata también. ¿Qué sucede con la primera actividad?

- En el peor caso, todas las demás actividades se superponen, con lo cual eligiendo la primer actividad, ya maximizamos $\mathcal{S}$ y obtenemos la solución óptima.
- En el caso que hubiese más actividades que no se superponen, sabemos que una combinación de ellas hacen una solución óptima O. Tomemos entonces la solución óptima $O = \{O_1, \ldots, O_m\}$, $m \leq n$.
- Como O está ordenada, entonces si $A_1 = O_1$ no hay nada que probar. si $A_1 \neq O_1$ sabemos que A_1 finaliza antes o a lo sumo a la misma hora t que O_1. Podemos reemplazar entonces A_1 por O_1 y observar que sigue siendo una solución válida, ya que A_1 no se superpone con el resto de las actividades en O (ya que O_1 no lo hacía). Llamemos O' a esta solución. Como O ya era de cardinalidad máxima y $\#O = \#O'$, entonces O' también es de cardinalidad máxima y por lo tanto sigue siendo óptima.

[1] En el pseudocódigo: $\bigoplus$ = concatenación.

- Haciendo esta observación, podemos entonces agregar A_1 a nuestra solución golosa ya que sabemos que no nos va a dar una solución errónea.

¿Qué sucede con las actividades restantes?

Podemos considerar todas las actividades que no se superponen con A_1 y analizarlo como un subproblema. Entonces, podemos realizar el mismo argumento anterior y elegir la actividad que termina más temprano, es decir la primer actividad de ese subproblema. Y con la misma idea seguimos para el resto de elementos.

Demostración por inducción

Propongamos un invariante y demostremos su correctitud por inducción:

$P(i)$: "B es una solución óptima del subproblema $\{A_1, \ldots, A_i\}$ y se puede extender a una solución óptima $C = \{C_1, \ldots, C_r, C_r + 1, \ldots, C_s\}$ con $s \leq n$. Escribimos $B = B_1, \ldots, B_r$ con $r \leq s$."

¿Vale $P(1)$?

Sí, vale $P(1)$ antes de entrar al ciclo ya que $B = \{A_1\}$ y argumentamos arriba que existe una solución óptima con A_1 como primer elemento.

¿Vale $P(i) \Rightarrow P(i+1)$?

Es decir, ¿Se mantiene el invariante en cada iteración? Supongamos por hipótesis inductiva que $B = B_1, \ldots, B_r$ es una solución óptima del subproblema $\{A_1, \ldots, A_i\}$ y se puede extender a una solución óptima C.

- Si A_{i+1} no se superpone con B_r, es decir $s_{i+1} < t_r$, entonces el algoritmo goloso ignora A_{i+1} y nos queda que B sigue siendo una solución óptima de $\{A_1, \ldots, A_{i+1}\}$. Además, como B se mantiene igual, en particular sigue siendo extensible a una solución óptima C.
- Si A_{i+1} no se superpone con B_r, es decir $s_{i+1} \geq t_r$, entonces nuestro algoritmo goloso incluye a A_{i+1} en la solución y nos queda $B' = B_1, \ldots, B_r, B_{r+1}$ donde $B_{r+1} = A_{i+1}$. Veamos entonces que sigue siendo una solución óptima para este subproblema y que sigue siendo extensible a una solución óptima C.
  - Como B era una solución óptima del subproblema $\{A_1, \ldots, A_i\}$ y A_{i+1} no entra en conflicto con ninguna A_j, $1 \leq j \leq i$, entonces B' es una solución óptima del subproblema $\{A_1, \ldots, A_{i+1}\}$ ya que sigue siendo de cardinalidad máxima.
  - Ahora, por hipótesis inductiva sabemos que $C = B_1, \ldots, B_r, C_{r+1}, \ldots, C_s$ es la extensión óptima de B. Entonces,
    - Si $B_{r+1} = C_{r+1}$, ya está, tenemos que $B_1, \ldots, B_r, B_{r+1}, C_{r+2}, \ldots, C_s$ es una extensión óptima de B'.
    - Si $B_{r+1} \neq C_{r+1}$ tenemos que ver si sigue existiendo una extensión óptima. Tomemos $C = B \bigoplus R$ donde $R = C_{r+1}, \ldots, C_s$. Sabemos que B_{r+1} no entra en conflicto con ninguna actividad de B y empieza después de todas ellas, con lo cual revisemos que pasa con R.
    - Como R está ordenado en orden ascendente, entonces B_{r+1} termina a lo sumo al mismo tiempo que C_{r+1} con lo cual puedo reemplazar C_{r+1} por B_{r+1}, el cual tampoco se superpone con el resto de actividades de R ya que C_{r+1} no lo hacía. Haciendo esto obtenemos $R' = B_{r+1}, C_{r+2}, \ldots, C_s$. Con el mismo argumento que el caso base, sabemos que R' sigue siendo una solución óptima del subproblema $\{A_{i+1}, \ldots, A_n\}$. Luego tenemos la extensión $C^* = B \bigoplus R' = B_1, \ldots, B_r, B_{r+1}, C_{r+2}, \ldots, C_s$ que es una extensión óptima de B'.

Al salir del ciclo, ¿Vale $P(n)$?

Sí, ya que el invariante nos garantiza que al salir del ciclo, B es una solución óptima del subproblema $\{A_1, \ldots, A_n\}$ y se puede extender a una solución óptima. Como $i = n$, tenemos que B es una solución óptima de $\mathcal{A} = \{A_1, \ldots, A_n\}$

4. Complejidad computacional

4.1. Cota teórica

Como vimos, el algoritmo primero ordena las actividades y luego las recorre una sola vez, haciendo operaciones de costo $\mathcal{O}(1)$ en cada iteración del ciclo. Es decir que el ciclo es de complejidad $\mathcal{O}(n)$, donde n es la cantidad de actividades. Por lo tanto, lo que tiene mas impacto en el algoritmo es el costo de ordenar las actividades. Si utilizamos la función *sort* que nos provee la *stdlib*, cuya complejidad es $\mathcal{O}(n\ log(n))$, tenemos que la complejidad del algoritmo es $\mathcal{O}(n + n\ log(n)) = \mathcal{O}(n\ log(n))$.

4.2. Experimentación

Vamos a mostrar empíricamente que el algoritmo tiene la complejidad mencionada. En primer lugar generamos aleatoriamente un conjunto de instancias de actividades con el que ejecutar nuestro algoritmo. Posteriormente hicimos la comparación de los tiempos de ejecución de nuestro algoritmo con la cota teórica estimada haciendo una regresión lineal sobre estos dos. Mostramos a continuación los gráficos obtenidos en escala normal y también en escala logarítmica para apreciar mejor los resultados:

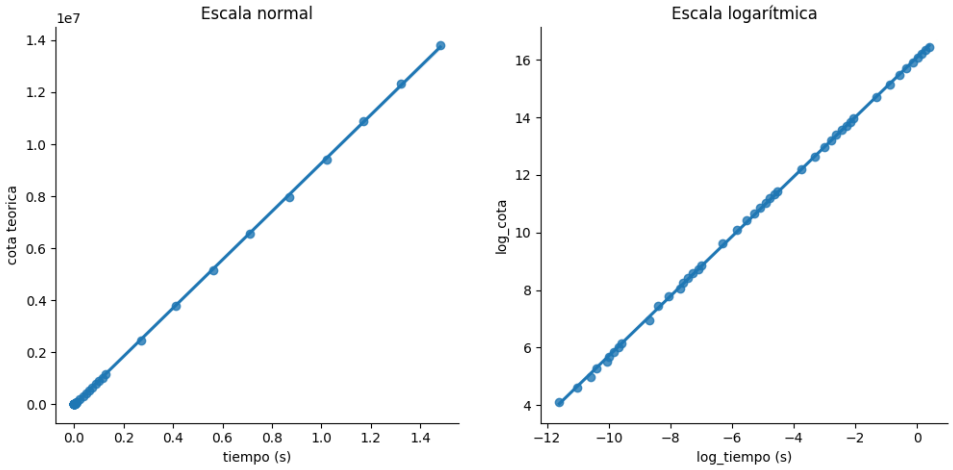

Figura 1: Regresión lineal de cota teórica con los tiempos obtenidos

Observando estos gráficos, podemos concluir con alto nivel de confianza que la cotas teóricas estimadas correlacionan con los tiempos de ejecución obtenidos.

Finalmente podemos analizar en qué casos este algoritmo funciona mejor o peor. Si consideramos tres instancias, una en la que las actividades ya se encuentran ordenadas ascendentemente, otra en la que también están ordenadas pero de manera descendente y otra de orden random, podemos observar lo siguiente:

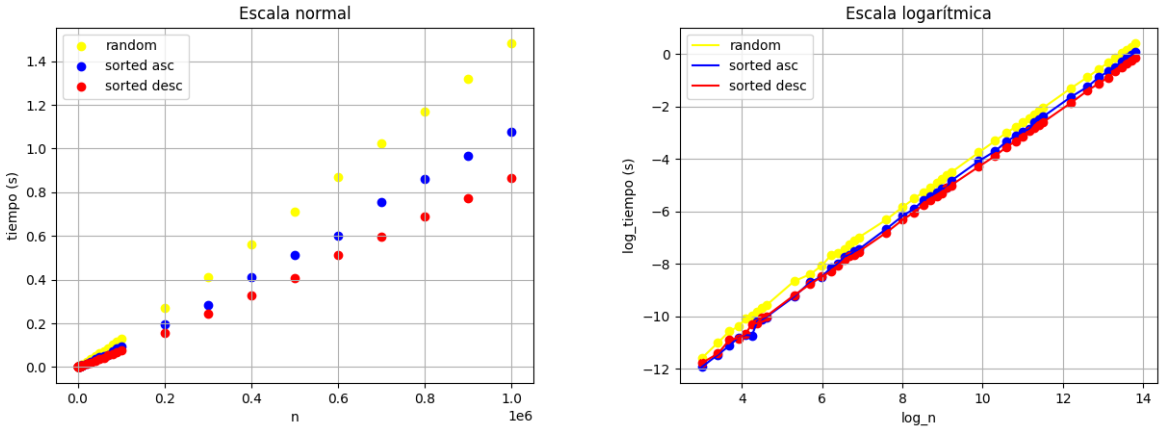

Figura 2: Comparación de instancias usando *std::sort*

Juntando los gráficos en uno podemos hacer la comparación entre familias de instancias y se puede observar que la ejecución de los casos ordenados de manera ascendente y descendente son más rápidos que el caso random. En particular se puede destacar que el más rápido es el caso ordenado de forma descendente lo cual nos lleva a concluir que utilizando la implementación de *std::sort*, el algoritmo funciona mejor cuando las instancias están ordenadas de esta manera.

Esto último nos llevó a plantear otro algoritmo de ordenamiento para incluir en la implementación de nuestro código, de esta forma sabremos si la implementación del *std::sort* fue gestora de los tiempos obtenidos. Recordemos que los tiempos de inicio y fin de las actividades están acotados por $2n$, con lo cual podemos implementar un *bucket sort* para ordenar las actividades. El resultado fue el siguiente:

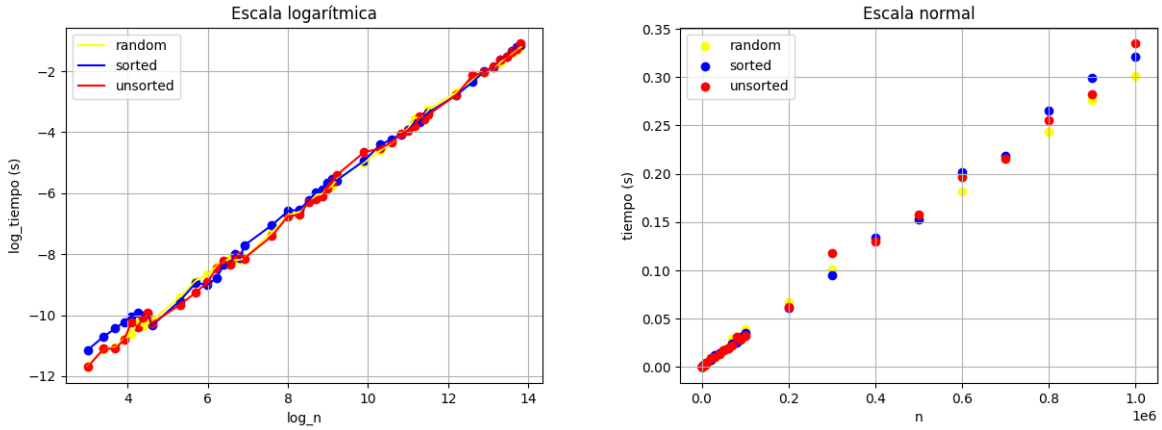

Figura 3: Comparando instancias usando *bucket sort*

A diferencia del sort anterior, con el *bucket sort* se asegura que la complejidad del algoritmo sea $\mathcal{O}(n)$, ya que crear los buckets y luego iterar por todas las actividades tiene costo lineal en la cantidad de actividades. Esto permite observar que los tiempos de ejecución de las distintas familias de instancias tienen una diferencia de tiempo bastante despreciable utilizando *bucket sort*.

En los siguientes gráficos mostramos la comparación de velocidad usando ambos algoritmos de ordenamiento:

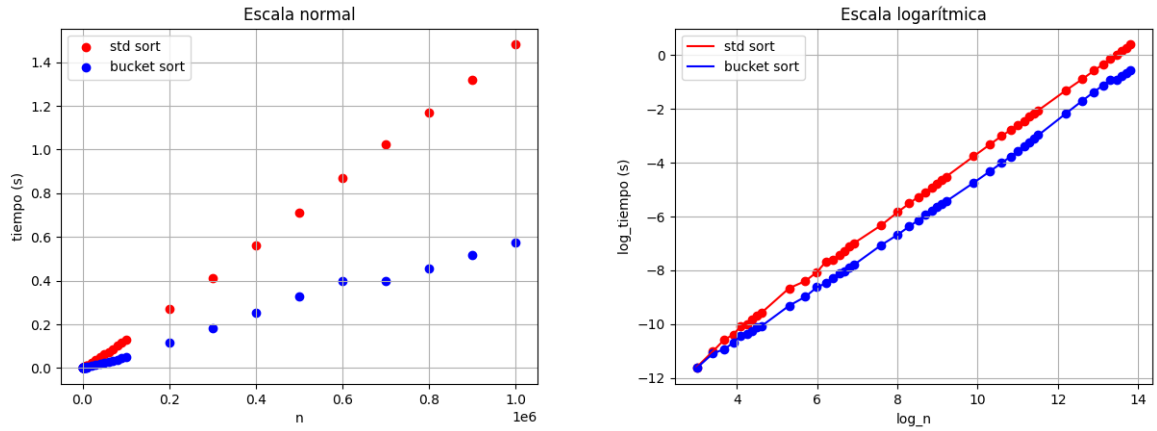

Figura 4: Comparación de *std::sort* con *bucket sort*

5. Conclusiones del experimento

Para finalizar, podemos concluir que si la cota es conocida (en este caso $2n$) entonces hacer un *bucket sort* mejorará la complejidad de nuestro algoritmo, pero hará que no haya un conjunto de instancias más fácil de resolver que otras. Por el contrario, si no hay una cota explícitamente establecida no sería conveniente usar *bucket sort* (ya que no sabemos cuál es el tamaño de buckets que deben hacerse y habría que recorrer todas las actividades para saberlo y posteriormente crearlo). Esto nos lleva a usar el *std::sort* que deja la complejidad del algoritmo en $\mathcal{O}(n\ log(n))$, donde se pudo observar que los conjuntos de instancias más fáciles de resolver eran las ordenadas, tanto ascendentes como descendentes (siendo estas últimas las de menor tiempo de ejecución) y las instancias más difíciles de resolver eran las que no están ordenadas.