

Algoritmos y Estructuras de Datos III

Trabajo Práctico 1

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

Algoritmo greedy Informe

Integrante	LU	Correo electrónico
Braginski Maguitman, Leonel Alan	385/21	leobraginski@gmail.com
Deganis, Gaston Lucas	295/20	gastondeganis@gmail.com

Parte I

Problemática

1. Actividades

Llamamos actividad a la tupla (s_i, t_i) donde $s_i, t_i \in \mathbb{N}$. s_i representa el inicio de una actividad y t_i el final de la misma. Una actividad es un intervalo en la recta \mathbb{N} .

Definimos $A = \{A_1, \dots, A_n\}$ como un conjunto de actividades. En particular para este problema suponemos que $1 \leq s_i < t_i \leq 2n$ para toda actividad en A

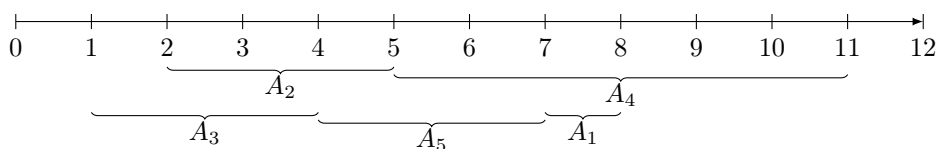
2. Problema

Dado A el conjunto de actividades previamente mencionado se busca diseñar un algoritmo que determine el subconjunto de actividades S que contenga la máxima cantidad de actividades de forma que no se solapen en la recta \mathbb{N} .

3. Ejemplo

Supongamos que tenemos el conjunto A el cual contiene las siguientes actividades

Actividad	s_i	t_i
1	7	8
2	2	5
3	1	4
4	5	11
5	4	7



Se puede ver como en este caso la solución óptima va a ser tomar $\{A_3, A_5, A_1\}$. Ya que no existe otra solución con cardinalidad mayor que cubra más o lo mismo que esta sin que se solape ningún intervalo.

Parte II

Algoritmo

Este problema se puede resolver usando una estrategia greedy en $O(n)$.

Esta solución se construye mediante un ciclo sobre los elementos del conjunto, en el cual se toma en cada iteración la actividad cuyo final t_i sea el más chico posible, y que no solape con las actividades previamente elegidas.

A simple vista esta implementación es $O(n^2)$ ya que por cada actividad busca el mínimo entre todas las actividades. Esto se puede mejorar si ya tenemos las actividades ordenadas sobre t_i . En particular para este problema sabemos que $1 \leq s_i < t_i \leq 2n$ por lo que podemos ordenar el conjunto de actividades en $O(2n)$ porque sabemos que t_i esta acotado en $2n$ y podemos utilizar CountingSort.

Tener las actividades ordenadas nos permite determinar en $O(n)$ el conjunto óptimo viendo en cada iteración i que A_i no solape con las actividades anteriores.

Es importante destacar que la solución óptima siempre va a tener incluida la primer actividad del conjunto ordenado. Esto se debe a que la primer actividad es siempre la del final mas chico posible, lo mismo que se busca por estrategia greedy.

maximizarActividades (in $A : \text{conj}(\text{Actividad}) \rightarrow P : \text{secu}(\text{Actividad})$)

```

1:  $Ord : \text{secu}(\text{Actividad}) \leftarrow \text{CountingSort}(A)$   $\triangleright O(2n)$ 
2:  $P \leftarrow \{ Ord_1 \}$ 
3:  $ult : \text{Actividad} \leftarrow Ord_1$ 
4: for  $2 \leq i \leq n$  do  $\triangleright O(n)$ 
5:   if  $Ord_i.s \geq ult.t$  then
6:      $P \leftarrow P \cup \{Ord_i\}$ 
7:      $ult \leftarrow Ord_i$ 
8: return  $P.indices$ 
```

Complejidad: $O(2n) + O(n) = O(n)$

4. Ultimas observaciones

Este algoritmo tiene complejidad lineal en todos sus casos, no existen instancias más fáciles o difíciles. Esto es porque en cualquier caso la entrada este ordenada o no, se ejecutara **CountingSort** y **maximizarActividades** con la misma complejidad para cualquier instancia. **CountingSort** no puede suponer que una instancia ya esta ordenada. Ordenar una instancia ya ordenada sigue siendo $O(2n)$

Parte III

Demostración

Aclaración: *Ult* es la última actividad válida agregada a *P*, *A* es el conjunto de actividades ya ordenado

Para demostrar que este algoritmo **greedy** es correcto y óptimo, vamos a definir un invariante *Q* que se cumplirá para cada iteración *i* :

$Q(i)$: "Los índices de *P* forman una secuencia de cardinalidad máxima de intervalos sin solaparse de A_1 a A_i "

5. Correctitud

Lo vamos a probar por inducción:

5.1. Caso Base:

Por definición del algoritmo se ordenarán las actividades de forma ascendente según el horario de terminación de la actividad (del *t*), y siempre *P* contendrá al primer elemento de ese conjunto ordenado.

Probemos entonces que vale $Q(1)$:

$Q(1)$: *P* tiene únicamente el índice de la primer actividad del conjunto ordenado, se puede deducir entonces que $A_i = A_1$
 \Rightarrow *P* cubre de A_1 a A_i
 \Rightarrow las actividades en *P* no se solapan, ya que hay una sola
 \Rightarrow *P* es un conjunto de cardinalidad máxima porque tienen todas las actividades de A_1 a A_i
 \therefore Vale el invariante en el caso base. Vale $Q(1)$

5.2. Paso Inductivo:

Suponemos que $Q(i)$ vale y queremos probar que también vale $Q(i + 1)$. Antes de la iteración $i + 1$, por HI, sabemos que *P* es el conjunto de máxima cardinalidad de intervalos sin solaparse de A_1 a A_i .

HI: La secuencia P tiene a los índices de las actividades que cubren de A_1 a A_i de máxima cardinalidad.

Al finalizar la iteración $i + 1$ se tendrá un conjunto P' que será una extensión de P según las decisiones que se tomen durante la iteración. Es decir: $P \subseteq P'$. Se tienen dos posibles decisiones:

5.2.1. Decisión 1

$$A_{i+1}.s < Ult.t$$

En este caso, el horario de inicio de la actividad del $i + 1$ -ésimo elemento ordenado ($A_{i+1}.s$) es menor estricto que el horario de finalización de la última actividad seleccionada ($Ult.t$). Es decir, se solapa. Por definición del invariante (y del **greedy**) no se pueden seleccionar dos actividades que se solapen. Acá $P = P'$

\Rightarrow Ult se solapa con P y no se agrega a P'

\Rightarrow P' cubre todos los puntos de A_1 a A_{i+1} , ya que son los mismos que P

\Rightarrow P' tiene cardinalidad máxima para una solución válida, ya que es la misma que P , la cual por HI sabemos que es máxima.

\therefore Podemos ver entonces que vale $Q(i + 1)$.

5.2.2. Decisión 2

$$A_{i+1}.s \geq Ult.t$$

Es decir, que el horario de inicio de la actividad del $i + 1$ -ésimo elemento ordenado ($A_{i+1}.s$) es mayor o igual que el horario de finalización de la última actividad seleccionada ($Ult.t$). En este caso, hay que agregar A_{i+1} a P .

Acá $P' = P \cup \{A_{i+1}\}$.

\Rightarrow $P \subseteq P'$ y por HI P cubre de A_1 a $A_i \Rightarrow$ al agregar A_{i+1} a P hace que P' cubra de A_1 a A_{i+1} .

\Rightarrow El tiempo de inicio de A_{i+1} es después de la finalización del último agregado a $P \Rightarrow P'$ no tiene actividades que se solapen.

\Rightarrow Por HI P es de cardinalidad máxima y P' extiende a P entonces P' también es de cardinalidad máxima.

\therefore Podemos ver entonces que vale $Q(i + 1)$

6. Optimalidad

Vamos a demostrar optimalidad usando el mismo invariante. Sabemos que la solución óptima P es aquella que cubre la mayor cantidad de actividades sin que ninguna se solape.

Tomamos $A = \{A_1, \dots, A_n\}$.

$Q(n)$ = "Los índices de P forman una secuencia de cardinalidad máxima de intervalos sin solaparse de A_1 a A_n "

Ahora bien, previamente demostramos que el invariante es correcto. $Q(n)$ vale al finalizar todas las iteraciones y nos asegura que P tiene la secuencia de índices de la solución óptima.

$\therefore Q(n) \checkmark \Rightarrow P$ es solución óptima

Parte IV

Demostración empírica

7. Experimento

Se probó el algoritmo con 16 conjuntos de datos arbitrariamente grandes. Esto con el objetivo de mostrar que la complejidad es $O(n)$ con pruebas reales. Para cada conjunto de pruebas generado probamos cuanto tardó el algoritmo con el cronometro de C++, estas pruebas se hicieron 10 veces por cada conjunto y sacamos un promedio de cada una. En 1* se muestran los tiempos de ejecución obtenidos en función del tamaño de los casos de test. En 2* usamos el

método de regresión lineal para formar una recta entre todos los puntos obtenidos. Esto muestra que la curva de complejidad es lineal.

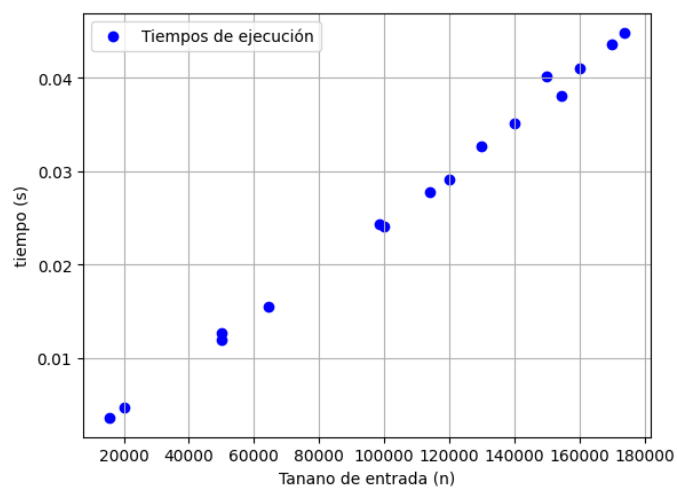


Figura 1: Tiempos obtenidos tras las pruebas

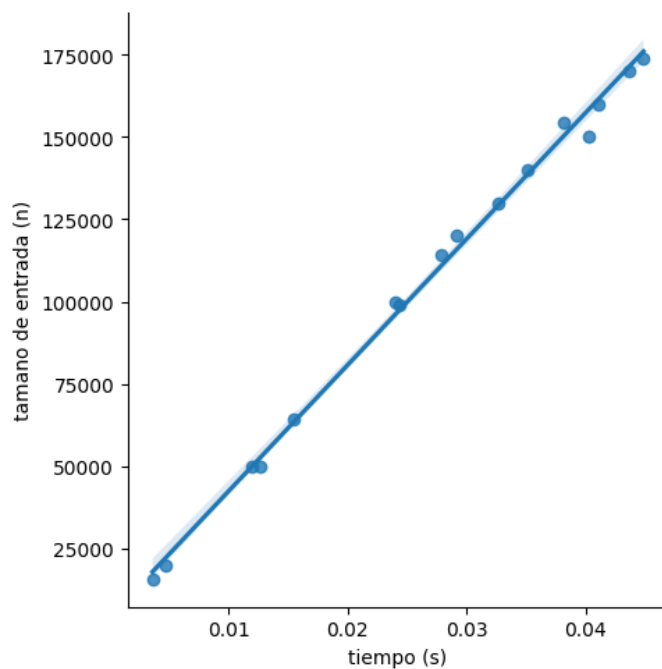


Figura 2: Regresión lineal aplicada a los datos obtenidos