Tarea 4 – Análisis de algoritmos

Vargas Aldaco Alejandro

Ejercicio teórico práctico:

El archivo es un proyecto de NetBeans (versión 8.2).

Precondiciones: REQ = {req_1, req_2, . . . , req_n} conjunto de n ≥ 1 tareas, donde cada tarea reqi tiene una descripción con un identificador y un intervalo constituido por su tiempo de inicio start(i) y su tiempo final end(i), con start(i) < end(i), y REQ está ordenado en forma no decreciente por tiempo de inicio.

Postcondiciones: Un arreglo SCH de n listas que representa la calendarización de REQ tal que

a) Para toda tarea reqi donde i = 1, . . . , n existe un j ∈ {1, . . . , min} tal que la lista SCH[j] contiene a req_i.

b) Cualesquiera dos tareas req_i y req_k en una lista SCH[j] con 1 ≤ j ≤ min son compatibles, es decir, end(k) ≤ start(i) o end(i) ≤ start(k).

c) min es el número mínimo de listas no vacías para las cuales se puede garantizar el cumplimiento de las dos condiciones anteriores.

Tomamos REQ como el conjunto de tareas de tamaño n = 10k, donde k=1,…,100, cuyas tareas está ordenado en forma no decreciente por tiempo inicial. Ésto es req_1.getStart() <= req_2.getStart() <= … <= REQ.get(REQ.size()-1).getStart(). Por otro lado tomamos SCH como el conjunto que contiene SCH_j = 1,…,min tal que para toda req_i existe SCH_j que contiene a req_i.

Para el caso donde sólo hay una tarea en REQ antes de ejecutar el algoritmo secheduleAll(REQ). Tomanos SCH de tamaño 1 y metemos a req_1 en SCH_1.

* Como REQ está ordenado en forma no decreciente, cuando REQ.size() > 1, veamos que si existe req_i ={ req_1, …,req_z} tal que req_1.getStart() = … = req_z.getStart(), para este paso SCH.size() = z dado que ningún elemento es compatible con otro. Ahora tomamos z+1, como z+1 no pertenece al conjunto anterior, entonces req_{z+1}.getStart() > req_z.getStart(), por lo cual existen dos casos:

** req_{z+1} es compatible con algún SCH_j.get(SCH.size()-1), con 1 <= j <= z, entonces SCH.size() = z;

*** req_{z+1} no es compatible con ningún SCH_j.get(SCH.size()-1), con 1 <= j <= z, como REQ esta ordenado en forma no decreciente por tiempo inicial, veamos que no hay req_i ={ req_1,…,req_z} para todo SCH_j.get(SCH.size()-1) , con 1 <= j <= z, tal que req_i.getEnd() <= req_{z+1}.getStart(), en este caso no queda más que crecer en uno SCH, por lo tanto SCH.size() = z+1 y req_{z+1} estará contenido en $SCH_{(z+1)-1}$

Análogamente a z+1 lo podemos hacer para w, con z+1 <= w <= n. Por lo cual se cumple b) por ** y ***, como sólo crecemos en uno para los casos * y *** se cumple c) y por *,** y *** se cumple a).

Veamos la complejidad del algoritmo. Como REQ es de tamaño n.

Para el peor de los casos en que ninguna tarea sea compatible → SCH.size() = n, → $O(n^2)$ dado que para cada req_i = {1,…,n} no existe SCH_j.get(SCH.size()-1), con 1 <= j <= n-1 tal que SCH_j.get(SCH.size()-1).getEnd() <= req_i.getStart().

Por otro lado, veamos que para cada req_i = {1,…,n} tenemos dos casos:

1) No existe SCH_j.get(SCH.size()-1), con 1 <= j <= i-1 tal que SCH_j.get(SCH.size()-1).getEnd() <= req_i.getStart().

2) Existe SCH_j.get(SCH.size()-1), con 1 <= j <= i-1 tal que SCH_j.get(SCH.size()-1).getEnd() <= req_i.getStart(). Y j es el primer elemento de los que cumplen la condición.

Entonces la complejidad del algoritmo es menor n^2 y mayor a n. Es decir O(nlog n).