DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1: Técnicas Algorítmicas

Primer cuatrimestre 2023 — Algoritmos y estructuras de datos III

Integrante	LU	Correo electrónico
Nombre apellido	LU	email
Lautaro Diaz Bonelli	633/20	lautaro.diazbonelli@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
https://exactas.uba.ar

1. Ejercicio 3

En este ejercicio tenemos un cojunto de actividades con un comienzo s_i y un final t_i que no viene previamente ordenado. Y nos piden deovler el subconjunto de activiades tal que no se solapen.
Vamos a resolverlo usando tecnicas de algoritmos golosos. Este algoritmo goloso va a elegir la actividad cuyo momento final sea lo más temprano posible, de entre todas las actividades que no se solapen con las actividades ya elegidas.

Vamos a demostrar por inducción que este algoritmo goloso es correcto:
Sea la solución nuestra usando un algoritmo goloso G=$g_1, g_2, ..., g_m$ y la solución óptima hipotética O=$o_1, o_2, \ldots, o_n$ con n $\geq$ m (ya que la solución óptima tendrá la misma o más cantidad de elementos que la nuestra) queremos probar que n = m.

Y sea F() la función que devuelve el tiempo final de cada actividad tal que F(g_1) <F (g_2) y F(o_1) < F(o_2). En general F(o_i) < F(o_j) para todo i < j. Eso quiere decir que tanto en G como en O, todos los elementos serán compatibles,y no habrá superposición de actividades. Del mismo modo vamos a definir a la función S() que devuelve el tiempo inicial de cada actividad.

Lema: i $\leq$ j => F(g_i) $\leq$ F(o_j)

Vamos a probar eso por inducción:

Caso base: con i=1:
F(g_1) == F(o_1) ya que el algoritmo goloso elige siempre primero al mínimo F de todo el arreglo de input.

Hipótesis Inductiva: F(g_i) $\leq$ F(o_i)

Quiero Probar Que: F(g_i) $\leq$ F(o_i) => F(g_{i+1}) $\leq$ F(o_{i+1})
Cuando i= i+1, F(o_{i+1}) va a ser el menor de todas las actividades restante
($\nexists a : actividad \in O/F(a) < F(o_{i+1})$) y S($o_{i+1}$) > F($o_i$) ya que no se solapan. Como S(o_{i+1}) > F(o_i) y F(o_i) $\geq$ F(g_i) (por hipótesis inductiva) S(o_{i+1}) > F(g_i) pero si eso ocurre entonces significa que o_{i+1} es compatible con G. Por lo que al elegir g_{i+1}, esta actividad no va a terminar después que o_{i+1}, en otras palabras F(g_{i+1}) $\leq$ F(o_{i+1}). Esto implica que G tiene al menos la misma cantidad de actividades que O ya que la última actividad de G termina antes o al mismo tiempo que la última de O. Pero si O tuviera una n-esima actividad posible más, esta actividad también sería compatible con G por lo que el algoritmo goloso ya la hubiera incluido.

La implementación del algoritmo goloso que utilizamos es con uso de la idea mencionada previamente de elegir la actividad cuyo momento final sea lo más temprano posible, de entre todas las actividades que no se solapen con las actividades ya elegidas.
Para ello primero ordenamos la lista de actividades por su tiempo final en O(n) y después creamos una lista vacía que vamos a usar de respuesta y recorremos la lista de actividades ya ordenada O(n) y si las actividades no se solapan con la lista de respuesta, las agregamos a esta.
Checkear que una actividad sea compatible con la lista de respuesta 'res' se puede hacer en O(1) ya que si la lista 'res' está vacía es trivial, y si no, no nos interesa checkear toda la lista, si no que solo nos interesa el valor final de la última actividad agregada a 'res' y el valor inicial de la actividad a agregar.
El ordenamiento se puede hacer en O(n) ya que el tiempo de inicio y fin de las actividades está acotado por 2*n, sin ese detalle, la única opción sería hacerlo en O(n*log(n)). Gracias a que están acotados podemos realizar un Bucket Sort en donde creamos un bucket por cada tiempo final de cada actividad, y como 2 actividades distintas pueden tener el mismo tiempo final, dentro del bucket ponemos una lista de iteradores (la posicion en donde se encuentra ese tiempo en la lista original para después recuperarlo en O(1)). Luego recorremos los buckets que se habrán ordenado

solos ya que ponemos cada tiempo en su número, por lo que al recorrer los buckets ya estará de menor a mayor, y también recorremos la lista de iteradores y vamos formando una nueva lista de iteradores de la lista original que devolveremos como respuesta. Todo el ordenamiento que se esta haciendo es en O(n) ya que la suma de cada bucket no vacío más todos los elementos en la lista de iteradores es igual a la cantidad de elementos que hay que ordenar.
La funcion de ordenamiento No devuelve una lista de iteradores a la lista original, si no que los iteradores son a la lista ordenada esto es asi para despues poder devolver no solo cuantas actividades no se solapan, si no que tambien cuales son dichas actividades ya que si no hacemos esto despues nos resultaria imposible saber que actividad es cual ya que al ordenarlo perdemos el orden original.

Una vez ya ordenados, obtener nuestra respuesta es tan facil como ir recorriendo la lista ordenada de iteradores, y quedarnos solo con los que nos se solapen.

Este algoritmo tiene complejidad O(n) ya que estamos ordenando la lista en O(n) y luego recorremos la lista de tamaño n para quedarnos solo con las actividades que no se solapen. Checkear si 2 actividades no se solapan es O(1)

A continuación ponemos la tabla con los tiempo que le llevo al algoritmo ejecutado en nuestra computadora resolver test proporcionados por la catedra en el 1er cuatrimestre de 2022 que habia un ejercicio muy similar:

Instancia	Tamaño	Tiempo (seg)
interval_instance_1	5	0
interval_instance_2	7	0
interval_instance_3	10	0
interval_instance_4	1	0
interval_instance_5	10^4	0.001
interval_instance_6	10^4	0.001
interval_instance_7	10^4	0.001
interval_instance_8	10^5	0.029
interval_instance_9	10^5	0.024
interval_instance_10	10^5	0.013
interval_instance_11	10^6	0.369
interval_instance_12	10^6	0.393
interval_instance_13	10^6	0.407

Cuadro 1: Tiempo calculado con un Ryzen 5

En el cuadro se puede ver claramante como cada vez que aumenta el tamaño de la entrada por 10, su tiempo tambien aumenta aproximadamente por 10. Lo que tiene sentido ya que el algoritmo es O(n).

Existe un conjunto de instancias que más fáciles de resolver, ya que la dificultad del algoritmo, y donde se pierde la mayor cantidad de tiempo, es a la hora de ordenar las actividades. Si estas ya viniesen ordenadas, a pesar de no modificar la complejidad, en la practica podriamos resolver el problema mucho mas rapido.
Por otro lado, tambien existe un conjunto de instancias que son más difíciles de resolver, ya que hay que considerar que si hubiese un n lo suficientemente grande para que no entre en memoria (o en su defecto que 2*n no entre), entonces de la manera que el algoritmo esta implementado no podriamos resolver esa instancia.
Sin contar con que nos quedemos sin memoria, no nos importa mucho como estan distribuidos los tiempos finales ya que estamos creando siempre n^2 buckets independientemente de la entrada. Lo unico que cambiaria seria si hay muchas actividades con el mismo tiempo final, entonces habria

un vector muy grande en uno de los buckets, en el caso contrario, si no hubiese empates, habria muchos buckets con un vector de 1 solo elemento, pero esto no deberia afectar el tiempo ni la memoria.