Victoria Klimkowski y Lorenzo Gandolfo

Resumen

Este informe detalla la resolución de un problema de optimización de elección de actividades utilizando un algoritmo goloso eficiente que otorga una solución O(n), es decir, complejidad lineal.

Introducción

Para este ejercicio tenemos un conjunto de actividades tales que sabemos el momento inicial y final de cada una.

Debemos encontrar la manera de elegir la mayor cantidad de actividades posible sin que se solapen en tiempo, asumiendo que podemos comenzar una nueva actividad en el momento exacto que terminó la anterior. Además, sabemos que el momento de inicio siempre es estrictamente menor al tiempo de finalización y ambos están acotados entre 0 y 2n donde n es la cantidad de actividades.

Más formalmente:

Dada una familia de intervalos $A = a_1, a_1, \ldots, a_n$ (las actividades) de la forma (s,t) con $s,t \in [0,2n]$ y $s<t$, debemos encontrar el subconjunto $S \subseteq A$ tal que S tenga la mayor cardinalidad posible y se pueda escribir a S como la unión de intervalos semiabiertos o abiertos. Cada intervalo $a \in A$ se representa con sus extremos izquierdo $s(a)$ y derecho $t(a)$, ($A = [s(a_i), t(a_i)]$) con $1 \leq i \leq n$.

Resolución

Hemos abordado la problemática a través de una estrategia algorítmica de tipo golosa, donde elegimos la actividad cuyo momento final sea lo más temprano posible, de entre todas las actividades que no se solapen con las actividades ya elegidas.

En la función **bucketSort**, comenzamos por construir una estructura de 2*n+1 buckets, en la que cada bucket representa un posible tiempo de finalización. A continuación, recorremos el vector de actividades original y asignamos a cada bucket correspondiente una tupla de la forma <*pos,*<*s,t*>>, donde "pos" indica la posición original del elemento, "*s*" representa el tiempo de inicio y "*t*" el de finalización. Es decir, cada elemento es guardado en el t-ésimo bucket. O(n)

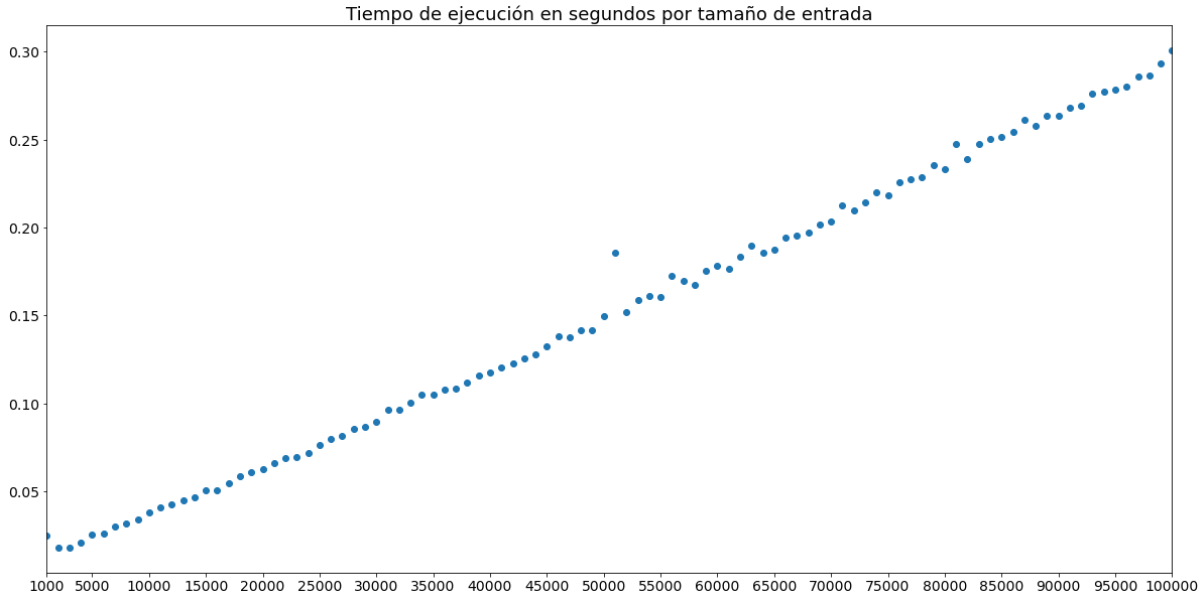
Cada bucket es un vector y se eliminan todos aquellos buckets de los 2*n+1 originales que estén vacíos, para luego devolver otro vector que contenga únicamente aquellos buckets que tengan al menos un elemento. O(n).

Acto seguido, se recorren las actividades ordenadas por tiempo de finalización, con el fin de seleccionar aquellas que se agregarán al vector resultado. La misma es elegida si su tiempo de comienzo es mayor o igual al tiempo de finalización de la actividad elegida anterior (y si hay más de una posible, tomamos arbitrariamente la que comience primero). O(n)

Una vez que ya tenemos armado el resultado, printeamos por pantalla solamente la primer componente de los elementos, es decir, la posición original.

Demostración empírica de la complejidad

Para efectivamente verificar que la complejidad es lineal corrimos 100 tests con tamaño entre 1000 y 100.000 y graficamos el tiempo que le tomó al algoritmo en cada tamaño. Los tests se realizaron utilizando python y las bibliotecas matplotlib, numpy y subprocess.



Es notoria la relación lineal entre el tamaño de entrada y el tiempo de ejecución (más allá de la presencia de outlayers). Es decir, vemos que la complejidad es O(n) empíricamente

Demostración golosa teórica

A es un subconjunto de actividades desde la posición 1 hasta i tal que no se solapan y están contenidas en una solución óptima S_{opt} y es maximal.

P(0): El conjunto vacío siempre estará incluído en la solución óptima y como no había actividades es maximal.

Asumamos que vale P(i) y queremos probarlo para P(i+1).

Tomamos A = $\{<pos_1, <s_1, t_1>>, \ldots, <pos_k, <s_k, t_k>>\} \subset S_{opt}$ las elegidas hasta la iteración i. Queremos ver qué pasa en la iteración i+1.

Si se solapan, $t_k \leq t_{i+1}$ porque estaban ordenadas pero $s_{i+1} < t_k$. Aquí no podríamos agregar $<pos_k, <s_k, t_k>>$ pues se solaparía con la anterior. Luego A queda igual que antes: $A \subset S_{opt}$ y es maximal.

Si no se solapan $t_k \leq t_{i+1}$ pero $s_{i+1} \geq t_k$. Sabemos que existe algún $<pos_j, <s_j, t_j>>$ tal que $t_j \geq t_{i+1}$. Si $t_{i+1} = t_j$ se puede agregar cualquiera ($<pos_j, <s_j, t_j>>$ o $<pos_{i+1}, <s_{i+1}, t_{i+1}>>$). Pero si $t_{i+1} < t_j$ agregar $<pos_j, <s_j, t_j>>$ no sería óptimo pues podría agregar $<pos_{i+1}, <s_{i+1}, t_{i+1}>>$ que deja más tiempo disponible.

Si agregase $<pos_j, <s_j, t_j>>$ no sería óptimo porque mi solución podría no ser maximal ya que como $t_{i+1} < t_j$ podría pasar que exista alguna $<pos_p, <s_p, t_p>>$ tal que $t_{i+1} \leq s_p < t_j$. Si pasa que $t_p \leq t_j$ claramente se sigue que $t_k \leq s_j \leq s_{i+1} \leq t_{i+1} \leq s_p < t_p \leq t_j$ o que $t_k \leq s_{i+1} \leq s_j \leq t_{i+1} \leq s_p < t_p \leq t_j$ y podríamos tener dos actividades ($<pos_{i+1}, <s_{i+1}, t_{i+1}>>$ y $<pos_p, <s_p, t_p>>$) en nuestra elección en lugar de una. El razonamiento para $t_j \leq t_p$ es análogo.

Luego si le agrego $<pos_{i+1}, <s_{i+1}, t_{i+1}>>$ a A sigue estando incluído en una solución óptima S_{opt} y es maximal porque nos aseguramos que en el medio de $<pos_k, <s_k, t_k>>$ y $<pos_{i+1}, <s_{i+1}, t_{i+1}>>$ no podría haber más actividades.

Ejemplo

El siguiente ejemplo detalla el comportamiento del algoritmo:

actividades = [(1,2),(1,3),(3,5),(2,4),(3,4),(4,6)]]

buckets = $[[]_0,[]_1,...,[]_{2n+1}$]

Luego del primer for loop:

buckets = [[],[],[(0,(1,2))],[(1,(1,3))],[(3,(2,4),(4,(3,4))],[(2,(3,5))],[(5,(4,6))]]

Notar que el índice de cada bucket se corresponde con la hora de finalización de cada actividad dentro.

Luego del segundo for loop se eliminan los buckets vacíos y nos quedamos con un vector de buckets ordenados donde cada uno contiene horarios de finalización menores a los del siguiente bucket.

buckets = [(0,(1,2))],[(1,(1,3))],[(3,(2,4)),(4,(3,4))],[(2,(3,5))],[(5,(4,6))]].

La función activsMax recorre el bucket modificado y selecciona de cada uno una actividad que no se solape con el horario de finalización de la actividad anterior.

res = [(0,(1,2)),(3,(2,4)),(5,(4,6))].

Se printea cada primer elemento+1. Es decir, las posiciones indexadas a partir del 1 en vez de el 0.

Conclusiones

Las estrategias golosas suelen requerir un análisis menos exhaustivo que el resto de las estrategias algorítmicas tales como backtracking, programación dinámica, etc. Esto se debe a que al entender cómo es la solución minimal para un único caso es posible aplicar esa solución globalmente. El problema de las actividades se pudo resolver con un método tan simple como seleccionar el mínimo de las horas de finalización de cada actividad y ordenarlo así tal que no se solapen; no fue necesario plantear una estructura de memoria ni una lógica recursiva ya que con un simple bucket sort se pudo lograr. Además fue posible demostrar que esta estrategia *greedy* es efectiva para cualquier conjunto de actividades.
Si bien la complejidad del bucket sort es explícitamente lineal, la demostración empírica fue una acertada verificación que comprobó su linealidad.

En resumen, la combinación de la estrategia golosa y el uso de bucket sort resultó en una solución eficiente y simple para el problema planteado. Además, agregamos que las actividades podrían ya venir ordenadas por lo cual ordenarlas sería innecesario y habría que cambiar ligeramente la implementación para que no haga el bucketsort. Sin embargo, si se usara el mismo código no habría diferencias significativas ya que todas las actividades igualmente seguirían siendo visitadas por el algoritmo. La complejidad seguiría siendo O(n) pero en la práctica podría ser más efectivo. Además, especulábamos que para un mismo n (tamaño de actividades de entrada) tardaría menos si había muchos repetidos o si se solapaban bastante (ya que debería printear menos actividades) pero corrimos 100 tests de tamaño 40000 y vimos que no podíamos establecer una relación entre ambas variables. Las actividades fueron generadas aleatoriamente y la cantidad maximal varió entre 300 y 340.