Teoría de Algoritmos

(75.29) Curso Buchwald - Genender

Trabajo Práctico 1

6 de abril de 2024

Zielonka Axel
110310

Petean Marina
110564

Romano Nicolas
110830

1. La Nacion del Fuego

1.1. Análisis del problema

El problema pide minimizar la suma ponderada de los tiempos de finalización: $\sum_{i=0}^{n} f_i \cdot b_i$
Vamos a reescribir la ecuación para que quede algo en lo que podamos operar de mejor manera:

$$\sum_{i=0}^{n} f_i \cdot b_i$$

Multiplicamos por $\frac{t_i}{t_i}$

$$\sum_{i=0}^{n} f_i \cdot b_i \cdot \frac{t_i}{t_i}$$

Llamando $a_i = \frac{b_i}{t_i}$

$$\sum_{i=0}^{n} f_i \cdot t_i \cdot a_i$$

Por último, llamando $s_i = f_i \cdot t_i$

$$\sum_{i=0}^{n} s_i \cdot a_i$$

En cada iteración queda entonces un producto entre un número s_i, que se sabe que es positivo (ya que $t_i > 0, \forall i$ y un número $a_i \geq 0 \forall i$. Por lo que después de cada iteración, la sumatoria va a crecer o mantenerse igual (en el caso de que $b_i = 0$)

1.2. Algoritmo Greedy propuesto

¿Por qué nuestro algoritmo es greedy?

Aplicamos una regla sencilla que permite obtener un óptimo local para el estado actual, y la aplicamos de forma iterativa para llegar a un óptimo general.

Esto lo logramos ordenando el arreglo por $a = \frac{b}{t}$ de mayor a menor, lo que nos permite al recorrer el mismo obtener en cada iteración un óptimo local (es decir, el elemento que optimiza la suma ponderada en ese momento). Al recorrer de manera ordenada el arreglo de batallas y calcular la sumatoria utilizando el óptimo local, que es el elemento actual según nuestra regla sencilla, conseguimos llegar al óptimo global (la suma ponderada mínima).

1.3. Optimalidad

1.3.1. Demostración de optimalidad

Sea $a_i = \frac{b_i}{t_i}$. Definimos una inversión de dos elementos batalla[i] y batalla[j] dentro de un orden de batallas a toda $a_i < a_j$ tal que $i < j$.

Supongamos que tenemos un orden de batallas tal que la suma ponderada $\sum_{i=0}^{n} f_i \cdot b_i$ sea la óptima (es decir, la mínima) y que ése órden posee inversiones como la anteriormente definida. Al revertir esas inversiones, es decir, reducir la cantidad de inversiones del orden de batalla, nosotros no podemos empeorar la suma ponderada final, solo podrá ser mejor *(ver demostración)*. Entonces, al revertir todas las inversiones, nos queda un arreglo ordenado de mayor a menor por la relación $a = \frac{b}{t}$ que es más óptimo que el óptimo con inversiones que habíamos supuesto. Lo que nos lleva a que el orden óptimo con inversiones es un absurdo, y por lo tanto el orden óptimo es un arreglo sin inversiones.

¿Por qué al revertir la inversión no podemos empeorar la suma ponderada? Supongamos que tenemos solo dos batallas (t_0, b_0) y (t_1, b_1) ordenadas tal que $a_0 < a_1$. Queremos demostrar que $t_0 \cdot b_0 + b_1 \cdot (t_0 + t_1)$ es siempre mayor que $t_1 \cdot b_1 + b_0 \cdot (t_1 + t_0)$. En otras palabras, estamos poniendo a (t_1, b_1) como primera batalla y a (t_0, b_0) como segunda para demostrar que es mejor que el orden anterior. Para esto, nos basta demostrar con método directo que la inecuación cumple.

$$t_1 \cdot b_1 + b_0 \cdot (t_1 + t_0) < t_0 \cdot b_0 + b_1 \cdot (t_0 + t_1)$$

Aplicando distributiva

$$t_1 \cdot b_1 + b_0 \cdot t_1 + b_0 \cdot t_0 < t_0 \cdot b_0 + b_1 \cdot t_0 + b_1 \cdot t_1$$

Restando $t_1 \cdot b_1$ y $b_0 \cdot t_0$ de ambos lados

$$b_0 \cdot t_1 < b_1 \cdot t_0$$

$$\frac{b_0}{t_0} < \frac{b_1}{t_1}$$

Es decir: $a_0 < a_1$. Partimos de una hipótesis que supusimos como verdadera, lo que quiere decir que el que tenía mejor relación peso-duración era más óptimo que el otro.

¿Que pasa en el caso en que dos elementos i y j de la forma (ti, bi) y (tj, bj) respectivamente, tengan el mismo a? Ya demostramos que la solución óptima es con un arreglo ordenado de mayor a menor a, Planteamos ahora un caso genérico de dos elementos consecutivos de nuestro arreglo: (supongo f al anterior a i y j)

Primer caso (ubicando i antes que j):

$$fi = f + ti$$

$$fj = fi + tj = f + tj + ti$$

$$Sumaponderada(caso1) = ant + (f + ti) \cdot bi + (f + tj + ti) \cdot bj + sig$$

Segundo caso (ubicando j antes que i):

$$fj = f + tj$$

$$fi = fj + ti = f + tj + ti$$

$$Sumaponderada(caso2) = ant + (f + tj) \cdot bj + (f + tj + ti) \cdot bi + sig$$

Aclaremos que con 'ant' y 'sig' nos referimos a la acumulacion de elementos de la sumatoria anterior y siguiente a los que nos interesa analizar. Notar que el F del que ubico último en ambos casos es igual, y por lo tanto el orden de i y j no afecta a los F siguientes.

Ahora planteo la igualdad de ambas sumas ponderadas, a ver si llego a un absurdo:

$$ant + (f + ti) \cdot bi + (f + tj + ti) \cdot bj + sig = ant + (f + tj) \cdot bj + (f + tj + ti) \cdot bi + sig$$

$$(f + ti) \cdot bi + (f + tj + ti) \cdot bj = (f + tj) \cdot bj + (f + tj + ti) \cdot bi$$

$$f \cdot bi + (ti \cdot bi) + f \cdot bj + tj \cdot bj + ti \cdot bj = f \cdot bj + tj \cdot bj + f \cdot bi + tj \cdot bi + ti \cdot bi$$

Cancelamos términos iguales:

$$ti \cdot bj = tj \cdot bi$$

$$\frac{b_j}{t_j} = \frac{b_i}{t_i}$$

$$a_j = a_i$$

Por lo tanto, vemos que se cumple que la suma ponderada es la misma al intercambiar dos elementos consecutivos, solamente si tienen el mismo a, por lo tanto nuestra solución es óptima también para este caso.

1.3.2. Cómo afectan los valores de t_i y b_i a la optimalidad

La variabilidad de los valores t_i y b_i no afectan la optimalidad del algoritmo, pues estamos ordenando por una relación entre el peso y el tiempo que siempre resulta en darnos el orden óptimo. En términos generales lo que buscamos es tener ordenado de mayor a menor peso, y de menor a mayor tiempo de manera simultanea, es por esto que al realizar b/t y ordenarlo de mayor a menor, funciona en todos los casos, cosa que no pasaria ordenando primero por peso y luego por tiempo, en lugar de por la relación entre ambos. Por ejemplo, si la duración de una batalla (t) es extremadamente grande, teniendo un peso grande en promedio, y b/a nos da un numero muy pequeño en comparacion a las demás, esta batalla estará en las ultimas del orden, pues así ese tiempo muy grande aparecerá en menos términos. Si lo hicieramos ordenando por tiempos de menor a mayor y luego por pesos de menor a mayor, el peso nos estaría importando más a la hora de ordenar, cuando en realidad ese tiempo tan grande hará que la suma ponderada se vuelva mucho más grande. Pasaría lo mismo si ordenaramos primero por peso, y luego por tiempo y tuvieramos un tiempo pequeño en promedio y un peso extremadamente chico.

Por lo tanto, nuestro algoritmo es óptimo sin importar como varien t_i y b_i. Lo único en lo que esta variabilidad influirá es en que posición estará la $batalla_i$ en el orden.

2. Minimizar la suma ponderada

El objetivo del trabajo es encontrar un algoritmo que minimice la suma ponderada de los tiempos de finalización, y que dicho algoritmo sea Greedy. Además, es necesario calcular su complejidad temporal, y realizar pruebas con ejemplos propuestos por nosotros.

2.1. Algoritmo planteado

A continuación se muestra el código de solución del problema.

```
def minimizar_suma(batallas):

    batallas_indices = crear_arreglo_con_indice(batallas)
    batallas_ordenadas = sorted(batallas_indices, key=lambda batalla: -(batalla
    [0][1] / batalla[0][0]))
    f = []
    for i in range(len(batallas_ordenadas)):
        if i == 0:
            f.append(batallas_ordenadas[i][0][0])
        else:
            f.append(f[i - 1] + batallas_ordenadas[i][0][0])

    suma_ponderada = 0
    for i in range(len(batallas_ordenadas)):
        suma_ponderada += f[i] * batallas_ordenadas[i][0][1]

    return suma_ponderada, batallas_ordenadas

def crear_arreglo_con_indice(arr):
    arreglo_con_indice = []
    for indice, valor in enumerate(arr):
        arreglo_con_indice.append([valor, indice])
    return arreglo_con_indice
```

2.2. Complejidad

La complejidad de este algoritmo se puede calcular realizando la suma de los sub-procesos que suceden dentro de él. Lo primero que sucede es que se crea un nuevo arreglo *batallas indices* que tiene como elementos a los mismos que *batallas*, agregando el índice de cada batalla a cada

tupla. Crear un arreglo, y hacerle n asignaciones tiene complejidad de $O(n)$ siendo n la cantidad de batallas (el único propósito es poder mostrar el número de batalla al mostrar el orden óptimo).

Luego se crea un nuevo arreglo, *batallas ordenadas* resultado de ordenar *batallas* de mayor a menor según $\frac{b_i}{t_i}$, utilizando la función *sorted()* de Python. Este algoritmo tiene una complejidad de $O(nlog(n))$. A continuación se realiza la asignación de la felicidad (F_i) de cada batalla, y como es crear un arreglo, recorrerlo completo y asignarle n variables esto tiene una complejidad $O(n)$

Por último, se realiza el cálculo de la suma ponderada, que es la sumatoria de los productos de las felicidades de cada batalla sus respectivos pesos. Estamos recorriendo el arreglo y haciendo operaciones constantes, por lo tanto eealizar n este proceso de suma tiene complejidad $O(n)$

Entonces, como estos procesos no son en simultáneo (no hay estructuras anidadas una dentro de otra), sus complejidades se suman y prevalece la mayor: $O(n) + O(nlog(n)) + O(n) + O(n) = O(nlog(n))$

Por lo tanto, la complejidad del algoritmo propuesto es de $O(nlog(n))$ siendo n la cantidad de batallas.

2.2.1. Análisis del efecto en la complejidad de la variabilidad de los valores t_i y b_i

Puesto que la complejidad no depende de las variables t_i y b_i, si no de n, siendo n la cantidad de batallas, como varíen estas no altera en nada la complejidad.

2.2.2. Mediciones y gráficos

Tiempos de ejecución con respecto a cantidad de batallas
Podemos observar que el grafico sigue la forma de la función xlog(x), lo que corrobora la complejidad teórica indicada.

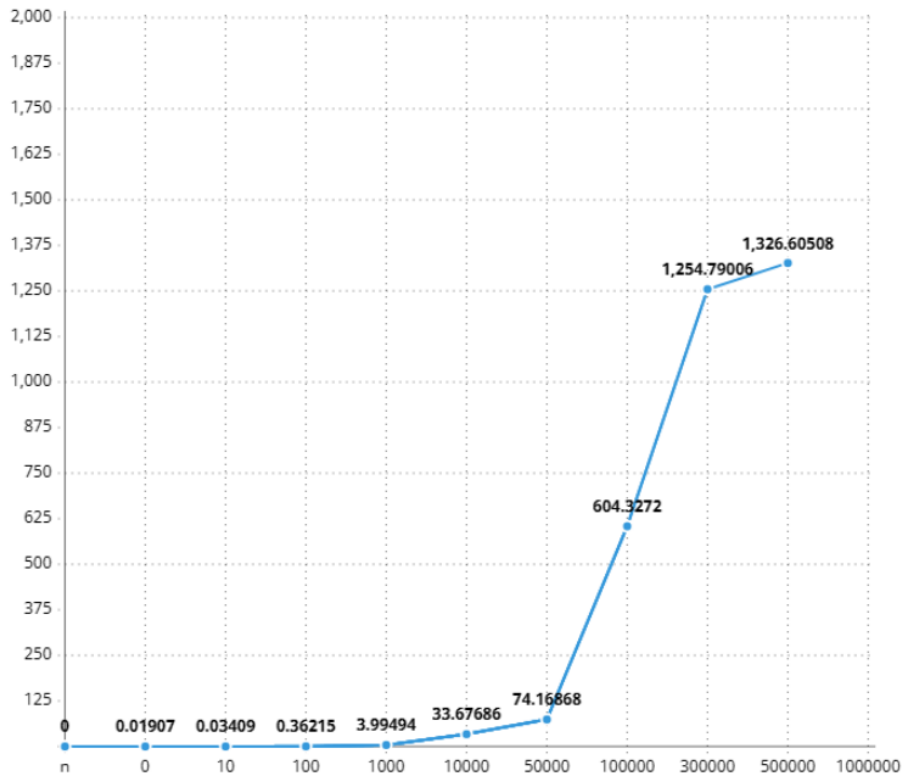

Figura 1: Gráfico de tiempos de ejecución (milisegundos) respecto a cantidad de batallas (n)

Tiempos de ejecución a igual N
Con este gráfico analizamos 5 posibles variaciones que pueden tomar t y b de las batallas y como varía el tiempo de ejecución del programa.

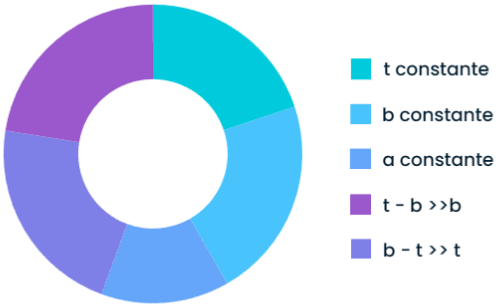

Figura 2: Gráfico circular de modos de variación

3. Testing

Para poder ver si nuestro algoritmo funcionaba correctamente, era necesario proveer de ejemplos de sets en donde variaban los tamaños, así como también los valores tanto del peso como del tiempo de cada batalla. En el directorio ***archivos*** del repositorio de Github se pueden encontrar además de los archivos provistos por la cátedra algunos archivos con distintos sets de batallas, que varían en tamaño y en los valores de los datos.

Por otro lado, en el directorio ***test*** del repositorio, está el archivo que se encarga de leer los ejemplos presentados por la cátedra y chequear que el resultado sea el esperado.

Donde los resultados esperados son:
Para n = 10: **309600**
Para n = 50: **5218700**
Para n = 100: **780025365**
Para n = 1000: **74329021942**
Para n = 5000: **1830026958236**
Para n = 10000: **7245315862869**
Para n = 100000: **728684685661017**

4. Conclusiones

Hemos desarrollado y análizado nuestro algoritmo Greedy propuesto para obtener el orden óptimo en el que realizar ciertas batallas para minimizar la suma ponderada de los tiempos de finalización más allá del set de datos que reciba. Hemos desarrollado sobre cómo selecciona el óptimo local en cada paso para lograr la solución óptima. Además, hemos estudiado la complejidad y cómo afecta la variabilidad de los datos a ésta y a los tiempos de ejecución y a la optimalidad del algoritmo. En conclusión, nuestro algoritmo Greedy óptimo brinda el órden en el que realizar las batallas que nos pidió El Señor del Fuego que logra minimizar dicha suma ponderada.