

TEORÍA DE ALGORITMOS
(75.29) CURSO BUCHWALD - GENENDER

Trabajo Práctico 1

Algoritmos Greedy



8 de abril de 2024

Santiago Varga
110561

Nahuel Cellini Rotunno
103320

Morena Sandroni
110205

1. Introducción

En este trabajo, presentaremos un algoritmo Greedy diseñado con el propósito de ayudar al Señor del Fuego a administrar sus batallas de forma tal que estas se resuelvan de la manera más efectiva posible. En este caso, tendremos en cuenta el tiempo de duración y la importancia de cada una de las batallas. Siendo t_i el tiempo que dura la batalla i y b_i su importancia, podemos definir F_i como el momento en el que termina cada batalla. Si la primera batalla es la j , entonces $F_j = t_j$. En cambio, si la batalla j se realiza justo después de la batalla i , entonces $F_j = F_i + t_j$. La efectividad radica entonces en minimizar la suma ponderada de los tiempos de finalización de las batallas en su totalidad, teniendo en cuenta también sus respectivos pesos (la importancia de cada una):

$$\sum_{i=0}^n b_i F_i$$

El algoritmo presentado debería dar una solución óptima para el problema dados los valores de n y todos los t y b . A lo largo de este trabajo, no solo analizaremos el funcionamiento del algoritmo, sino que también evaluaremos su complejidad temporal. Además, exploramos su capacidad de adaptación a diferentes escenarios, considerando variaciones en los tiempos y los pesos de las batallas. En las siguientes secciones, vamos a explorar como llegamos al algoritmo óptimo, teniendo en cuenta que atravesamos diferentes etapas antes de dar con la solución ideal. Analizaremos cómo funcionaban estos algoritmos, porque llegamos a ellos y como nos dimos cuenta que no eran suficientemente óptimos, evaluaremos su desempeño en varios contextos y, al final, sacaremos una conclusión general.

2. Análisis del problema

Para poder minimizar la suma ponderada de los tiempos de finalización de las batallas, incluyendo en esta su respectiva importancia, analizamos la posible relación existente entre los pesos y los tiempos de cada batalla y cómo estos modifican el resultado esperado.

Si analizamos los términos de la sumatoria podemos ver que F_i es siempre creciente, ya que en definitiva es la suma de todos los tiempos empleados en las diferentes batallas, mientras que b_i va variando y puede ser un número grande o chico. Como estos dos términos se multiplican entre sí, si queremos minimizar la sumatoria los términos tienen que ser lo más chicos posibles para que las adiciones sean lo más chicas posibles. Siguiendo esta lógica podríamos pensar que poniendo las batallas menos importantes al principio para tener términos chicos podría ser una opción, pero esto significa que las batallas más importantes son las que quedan para el final y esto va a hacer que los últimos términos que estemos sumando sean muy grandes por estar multiplicando entre sí dos números grandes. Entonces lo mejor es ordenar por batallas de mayor a menor importancia, lo que haría que los términos se equilibren, ya que a pesar de tener sumandos que pueden ser grandes al principio, los valores chicos de la importancia de las batallas van a hacer que los últimos valores de la sumatoria no sean tan grandes.

Esto se puede ver en el siguiente ejemplo:

Supongamos las siguientes batallas, siendo $B_i = (b_i, t_i)$:

- $B1 = (10, 3)$
- $B2 = (8, 15)$
- $B3 = (4, 2)$

Ordenando por importancia de menor a mayor:

$$B3 - B2 - B1 \rightarrow \sum_{i=0}^n b_i F_i = 4*2 + 8*17 + 10*20 = 8 + 136 + 200 = 344$$

Ordenando por importancia de mayor a menor:

$$B1 - B2 - B3 \rightarrow \sum_{i=0}^n b_i F_i = 10*3 + 8*17 + 4*20 = 30 + 136 + 80 = 246$$

Bajo esta línea de pensamiento también podríamos pensar que solo controlando el crecimiento de la finalización de la batalla se podría también minimizar la sumatoria. Pero si no tenemos en cuenta la importancia, por más que se controle ese número ordenando por tiempos de mayor a menor, no podemos garantizar que la batalla más importante no quede para el final y haga que el último sumando sea un número muy grande, ya que se estaría multiplicando el valor de importancia máxima con el valor de tiempo de finalización máximo (Esto se puede ver al comparar los últimos sumandos del ejemplo anterior, 200 es mucho mayor que 80). Ante esto también debemos preguntarnos: ¿qué pasa cuando hay batallas con la misma importancia? ¿es indistinto pelearlas en el orden que sea? ¿o debemos garantizar que se peleen unas antes que otras dependiendo del tiempo?

Para responder estas preguntas planteamos el siguiente ejemplo:

- $B1 = (10, 3)$
- $B2 = (8, 15)$
- $B3 = (10, 5)$

Ordenando por importancia de mayor a menor las dos opciones que tenemos son las siguientes:

$$B1 - B3 - B2 \rightarrow \sum_{i=0}^n b_i F_i = 10*3 + 10*8 + 8*23 = 30 + 80 + 184 = 294$$

$$B3 - B1 - B2 \rightarrow \sum_{i=0}^n b_i F_i = 10*5 + 10*8 + 8*23 = 50 + 80 + 184 = 314$$

Se puede observar que, a igual importancia, si se pelean las batallas más cortas primero se minimiza la sumatoria, ya que de esta forma, tomando solo los empates, el primer sumando aporta un valor más chico que si no se toma en cuenta los empates o si se prioriza la batalla más larga primero. Entonces deberíamos buscar un algoritmo que no solo priorice las batallas más importantes primero, sino que además tenga en cuenta las batallas de menor duración. ¿Existe una manera de relacionar ambos parámetros de forma tal que se resuelva de manera óptima el problema?

3. Algoritmo propuesto

Finalmente, tomando como referencia el caso del problema de la mochila, podríamos establecer una relación entre la importancia de cada batalla y su duración de la forma b_i/t_i y luego ordenar los resultados de estas relaciones de mayor a menor. De este modo, se priorizan las batallas de mayor importancia y, a su vez, las de menor duración.

Volviendo al ejemplo anterior:

- $B1 = (10, 3)$
- $B2 = (8, 15)$
- $B3 = (10, 5)$

Calculando los pesos segun la relacion b_i/t_i :

- $B1 = 10/3 = 3,33$
- $B2 = 8/15 = 0,533$
- $B3 = 10/5 = 2$

Ordenando de mayor a menor podemos garantizar que cuando se empata en la importancia de las batallas se prioricen las de menor tiempo:

$$B1 - B3 - B2 \rightarrow \sum_{i=0}^n b_i F_i = 10*3 + 10 * 8 + 8 * 23 = 30 + 80 + 184 = 294$$

Para ver si este camino es el correcto vamos a probarlo en el primer ejemplo:

- $B1 = (10, 3)$
- $B2 = (8, 15)$
- $B3 = (4, 2)$

La relación b_i/t_i es:

- $B1 = 10/3 = 3,33$
- $B2 = 8/15 = 0,533$
- $B3 = 4/2 = 2$

Ordenando de mayor a menor nos queda:

$$B1 - B3 - B2 \rightarrow \sum_{i=0}^n b_i F_i = 10*3 + 4*5 + 8 * 20 = 30 + 20 + 160 = 210$$

En comparación a la solución planteada anteriormente, con este algoritmo da una suma ponderada menor. Pero también debemos preguntarnos qué pasa cuando dos batallas tienen exactamente la misma relación.

Para esto planteamos entonces un tercer ejemplo:

- $B1 = (10, 5)$
- $B2 = (8, 15)$
- $B3 = (4, 2)$

La relación b_i/t_i es:

- $B1 = 10/5 = 2$
- $B2 = 8/15 = 0,533$
- $B3 = 4/2 = 2$

Siendo iguales sus relaciones, los dos ordenamientos posibles son:

$$B1 - B3 - B2 \rightarrow \sum_{i=0}^n b_i F_i = 10*5 + 4*7 + 8 * 22 = 50 + 28 + 176 = 254$$

$$B3 - B1 - B2 \rightarrow \sum_{i=0}^n b_i F_i = 4*2 + 10*7 + 8 * 22 = 8 + 70 + 176 = 254$$

Se puede observar que el orden entre B1 y B3 es irrelevante ya que la sumatoria da igual, por lo que podemos decir que cuando hay un empate entre relaciones no importa el orden. Entonces podríamos afirmar que el algoritmo planteado es el que ordena de mayor a menor según la relación b_i/t_i .

```
1 def ordenar_por_proporcion(batallas):
2     batallas.sort(key = lambda x: x[1]/x[0], reverse=True)
3     return batallas
4
5 def calcular_tiempo_total(batallas):
6     batallas = ordenar_por_proporcion(batallas)
7     coef_de_impacto = 0 # Suma acumulada de los tiempos de finalizacion
8     t_total = 0 # Tiempo total
9     for t_act, b_act in batallas:
10         coef_de_impacto += (t_total + t_act) * b_act
11         t_total += t_act
12     return coef_de_impacto
```

4. Complejidad Temporal

La complejidad del algoritmo es de $\mathcal{O}(n \log n + n)$, ya que es la complejidad del algoritmo de ordenamiento Timsort, que es el que utiliza la función `.sort()` de python, y la complejidad de recorrer la lista de elementos, haciendo operaciones $\mathcal{O}(1)$ en el medio. Como se cumple que $\mathcal{O}(n \log n) > \mathcal{O}(n)$, finalmente decimos que la complejidad del algoritmo es $\mathcal{O}(n \log n)$. En el gráfico se puede observar que el algoritmo tiene una tendencia a $\mathcal{O}(n \log n)$:

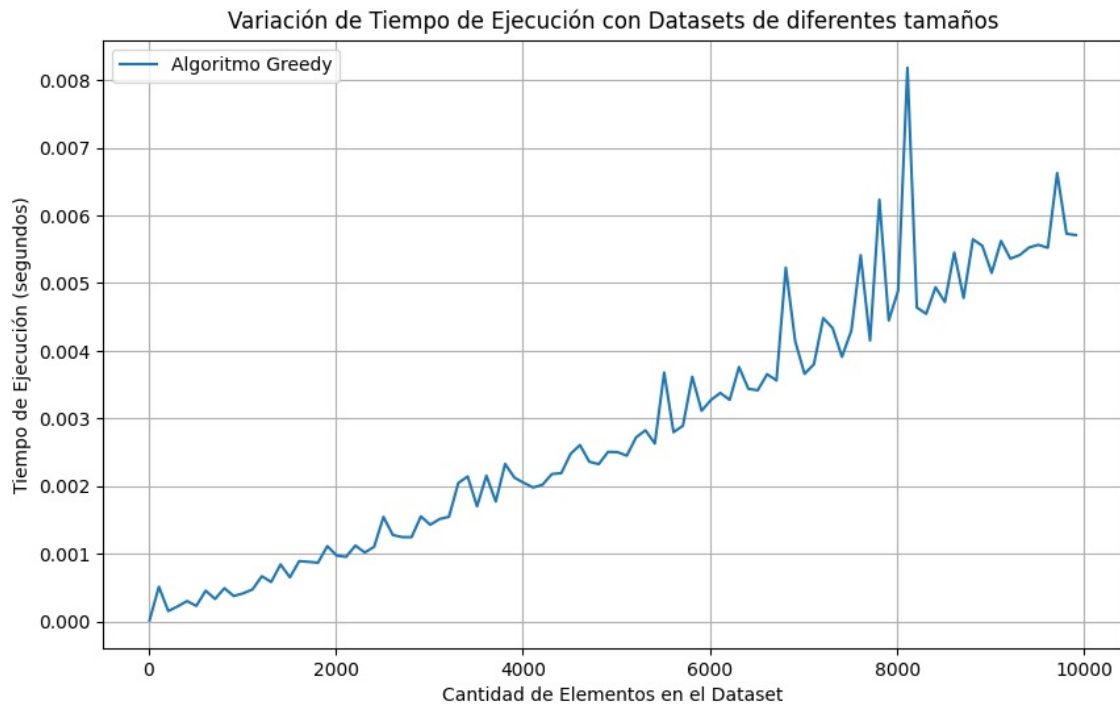


Grafico 1. Variación del tiempo de ejecución con diferentes tamaños de sets de datos.

5. Conclusión

Como conclusión, en este trabajo pudimos ver cómo los algoritmos Greedy pueden acercarse a soluciones óptimas en problemas de optimización de tiempo. A través de diferentes enfoques, buscamos ayudar al Señor del Fuego a administrar sus batallas de forma tal que estas se resuelvan de la manera más efectiva posible. Aunque creemos que el algoritmo basado en la relación b_i/t_i es óptimo, reconocemos que podría existir un algoritmo no Greedy que ofrezca mayor precisión a cambio de más tiempo de ejecución.