Teoría de Algoritmos

Curso Buchwald - Genender

Trabajo Práctico 1: Algoritmos Greedy en la Nación del Fuego

8 de abril de 2024

Candela Matelica	110641
Jesabel Pugliese	110860
Yamila Shih	110521

1. Introducción

El presente informe reune la documentación de la solución al problema propuesto para el primer trabajo práctico de la materia Teoría de Algoritmos, utilizando una metodología de resolución de tipo Greedy.

2. Presentación del problema

Es el año 10 AG, y somos asesores del Señor del Fuego (líder supremo de la Nación del Fuego). El Señor del Fuego tiene varias batallas con las cuales lidiar. Sabemos cuánto tiempo necesita el ejército para ganar cada una de las batallas (t_i). El ejército ataca todo junto, no puede ni conviene que se separen en grupos. Es decir, no participan de más de una batalla en simultáneo. La felicidad que produce saber que se logró una victoria depende del momento en el que ésta se obtenga (es decir, que la batalla termine). Es por esto que podemos definir a F_i como el momento en el que se termina la batalla i. 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 $F_j = F_i + t_j$

Además del tiempo que consume cada batalla, sabemos que al Señor del Fuego no le da lo mismo el orden en el que se realizan, porque comunicar la victoria a su nación en diferentes batallas genera menos impacto si pasa mucho tiempo. Además, cada batalla tiene una importancia diferente. Vamos a definir que tenemos un peso b_i que nos define cuán importante es una batalla.

Dadas estas características, se quiere buscar tener el orden de las batallas tales que se logre minimizar la suma ponderada de los tiempos de finalización: $\sum_{i=1}^{n} b_i F_i$.

El Señor del Fuego nos pide diseñar un algoritmo que determine aquel orden de las batallas que logre minimizar dicha suma ponderada.

3. Consigna

Se pide proponer un algoritmo Greedy que obtenga la solución óptima al problema planteado: Dados los n valores de todos los t_i y b_i, determinar cuál es el orden óptimo para realizar las batallas en el cuál se minimiza $\sum_{i=1}^{n} b_i F_i$.

4. Resolución

4.1. Análisis del problema

En esta ocasión, nos enfocaremos en ayudar al Señor del Fuego a analizar las batallas que enfrentará, con el objetivo de minimizar $\sum_{i=1}^{n} b_i F_i$. Para comenzar, definimos las variables clave:

- $n \rightarrow$ cantidad de batallas a combatir.
- $F_i \rightarrow$ momento en el que se termina la batalla i.
- $t_i \rightarrow$ tiempo que necesita el ejército para ganar la batalla i.
- $b_i \rightarrow$ qué tan importante es la batalla i.

Planteamos diferentes criterios y a medida que descartamos soluciones y detectamos patrones, nos encaminamos hacia nuestra solución final.

La pregunta clave es: ¿Cuál es el orden óptimo para combatir las batallas? Necesitamos determinar cuál será el próximo enfrentamiento una vez que concluyamos con uno.

A continuación, procedemos a realizar una demostración de nuestro método de análisis.

Dado que la sumatoria es de la forma $\sum_{i=1}^{n} b_i F_i$, nos interesaría minimizar o bien b_i, o bien t_i a mayor i, ya que sabemos que F_i va aumentando a mayor i. Analicemos qué pasa si nos enfocamos en minimizar cada uno de estos por separado:

¿Qué pasa si ordenamos de mayor a menor valor?

Por ejemplo, siendo (T_i, B_i) y tenemos 1° (20,15), 2° (11,10), 3° (9,8) (como podemos ver, ya estan ordenados de mayor a menor valor).

Sabemos que la mínima suma se da para el valor de 870. Si llevamos a cabo la sumatoria para el orden propuesto nos queda:

$\sum_{i=1}^{n} b_i F_i = 15 \times 20 + 10 \times (20 + 11) + 8 \times (20 + 11 + 9) = 15 \times 20 + 10 \times 31 + 8 \times 40 = 930$

$930 > 870$, por tanto el orden propuesto para las batallas no es el óptimo para minimizar la sumatoria.

A continuación, el código que utilizamos para calcular la sumatoria con las batallas ordenadas de mayor a menor valor:

```python
def orden_de_las_batallas_mayor_valor(batallas):
    batallas_ord = sorted(batallas, key=lambda x: x[2], reverse=True)  # Ordenar de mayor a menor valor
    F = 0
    suma_total = 0

    for i in range(len(batallas_ord)):
        F += batallas_ord[i][1]
        suma_total += F * batallas_ord[i][2]

    return batallas_ord, suma_total
```

¿Qué pasa si ordenamos de mayor a menor tiempo?

Dado que nos interesa minimizar la diferencia entre el final de una batalla y su comienzo, para así minimizar cada multiplicación $b_i F_i$ a mayor i, tendría sentido que los mayores tiempos de batalla los pongamos al comienzo para así minimizar las sumatoria $\sum_{i=1}^{n} b_i F_i$. Analicemos qué sucede si hacemos esto.

Ordenamos las batallas dadas en el punto anterior por mayor a menor tiempo de duración, quedándonos: 1° (20,15), 2° (11,10), 3° (9,8), el mismo orden que en el punto anterior, por lo que, por los cálculos ya hechos, sabemos que no es la mínima suma.

A continuación, el código que utilizamos para calcular la sumatoria con las batallas ordenadas de mayor a menor tiempo:

```python
def orden_de_las_batallas_mayor_valor(batallas):
    batallas_ord = sorted(batallas, key=lambda x: x[1], reverse=True)  # Ordenar de mayor a menor tiempo
    F = 0
    suma_total = 0

    for i in range(len(batallas_ord)):
        F += batallas_ord[i][1]
        suma_total += F * batallas_ord[i][2]

    return batallas_ord, suma_total
```

Estos resultados nos llevan a pensar en la importancia de la relación entre b_i y t_i para minimizar cada multiplicación $b_i F_i$ a mayor i. Por esto, probamos tomando las batallas teniendo en cuenta el orden de la relación entre la duración y la importancia de las mismas.

4.2. Algoritmo

A continuación se muestra el código de solución del problema.

```python
def orden_de_las_batallas(batallas):
    batallas_ord = sorted(batallas, key=lambda x: x[1] / x[2])
```

```
    F = 0
    suma_total = 0

    for i in range(len(batallas_ord)):
        F += batallas_ord[i][1]
        suma_total += F * batallas_ord[i][2]

    return batallas_ord, suma_total
```

Nuestro Algoritmo consiste en los siguientes pasos:

1. Primero, las batallas se ordenan según una proporción calculada: el tiempo de finalización de la batalla dividido por su importancia. Con esto, nos referimos a que el algoritmo priorizará las batallas con una proporción más baja, es decir, aquellas que terminen más rápido en relación con su importancia.

```
    batallas_ord = sorted(batallas, key=lambda x: x[1] / x[2])
```

2. Luego, inicializamos una variable F que representa el momento en el que se termina la última batalla en el orden actual, y una variable *suma total* para mantener el total de la suma ponderada.

3. De manera iterativa, se van recorriendo las batallas ordenadas. Para cada batalla, agrega su tiempo de finalización al tiempo acumulado F y luego suma el producto de F (el momento de finalización actual) y la importancia de la batalla a *suma total*.

```
    for i in range(len(batallas_ord)):
        F += batallas_ord[i][1]
        suma_total += F * batallas_ord[i][2]
```

4.3. Complejidad

Si tomamos n como la cantidad de batallas, entonces ordenar el arreglo de batallas es $\mathcal{O}(n \log n)$, luego recorrer este arreglo es $\mathcal{O}(n)$. Las operaciones que se realizan mientras se lo es recorrido son $\mathcal{O}(1)$, al igual que el resto de operaciones en el algoritmo (inicializar variables y luego devolverlas en la función). Por lo tanto, la complejidad queda:

$$\mathcal{O}(n \log n) + \mathcal{O}(1) + \mathcal{O}(n) \cdot \mathcal{O}(1) + \mathcal{O}(1) = \mathcal{O}(n \log n) + \mathcal{O}(n) = \mathcal{O}(n(\log n + 1)) = \mathcal{O}(n \log n)$$

La complejidad del algoritmo propuesto es $\mathcal{O}(n \log n)$.

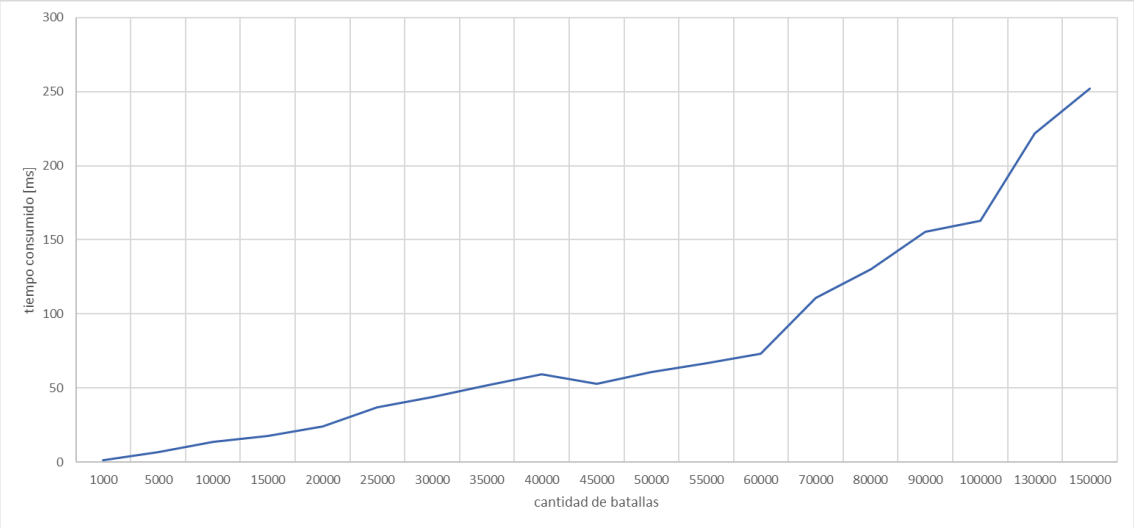
Por otro lado, dado que la complejidad depende únicamente de la cantidad de batallas n, la variabilidad de los valores de t_i y b_i no afecta a los tiempos del algoritmo planteado.

4.4. Optimalidad

Dados los vectores de tiempo $[t_1, t_2, ..., t_n]$ y valores de las batallas $[b_1, b_2, ..., b_n]$, sabemos que al ordenarlos de menor a mayor por su relación $\frac{t_i}{b_i}$ con $1 \leq i \leq n$ obtenemos la solución óptima al problema de minimizar $\sum_{i=1}^{n} b_i F_i$ ya que estamos colocando siempre al principio de la sumatoria a aquellas batallas que proporcionan más valor por unidad de tiempo, y dado que los valores de F_i dependen no solo del tiempo, sino de F_{i-1} (en caso de $i > 0$), estamos minimizando también la diferencia entre F_i y F_{i-1} a mayor i, por lo que al final resulta en la mayor reducción posible de la multiplicación $b_i F_i$ para cada i, y por tanto así del total de las sumas acumuladas. De esta forma, también podemos confirmar que la variabilidad de los valores de t_i y b_i no afecta a la optimalidad del algoritmo.

5. Mediciones

Se llevaron a cabo mediciones del tiempo de ejecución del algoritmo para distintas cantidades de batallas n, teniendo en cuenta distintos tipos de variabilidad entre la duración de cada una y sus valores, y se registraron los resultados en un gráfico para corroborar que se cumple la complejidad teórica del algoritmo.



6. Conclusiones

El presente trabajo nos permitió poner en práctica nuestros conocimientos sobre algoritmos Greedy e implementarlos en un algoritmo en que podamos ser capaces de visualizar su funcionamiento. Además nos permitió comparar varias opciones posibles de resolución del algoritmo y ser capaces de elegir la mejor, en base tanto a resultados experimentales como a lógica matemática.

Asimismo, realizamos análisis de optimalidad y complejidad que pudimos comprobar experimentalmente con pruebas hechas tanto por la cátedra como por el grupo. En particular, para la complejidad trabajamos con un set de pruebas cuyos valores de tiempos de duración de las batallas e importancias de las mismas variaban según distintos criterios para, de esta forma, poder contemplar bastantes casos y asegurarnos de que se cumpla la complejidad teórica calculada. Como podemos observar en el gráfico, la función efectivamente tiende a $\mathcal{O}(n \log n)$.