Teoría de algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 1
Algoritmos Greedy en la nación del fuego

---

8 de abril 2024

Integrantes:

- Matias Vazquez Morales (111083)
- Scarlet Mendoza (108524)
- Nestor Palavecino (108244)

1. Introducción

En este informe, presentaremos un enfoque basado en algoritmos greedy para resolver este problema complejo, proporcionando al Señor del Fuego una herramienta invaluable para optimizar su estrategia militar y asegurar el máximo impacto de sus victorias en la Nación del Fuego.

Además analizaremos:

- Complejidad
- Variabilidad de algunos valores en el algoritmo planteado
- Tiempos de ejecución para corroborar la complejidad teórica indicada
- Efectividad, para saber si la regla sencilla define una solución óptima
- Demostrar que siempre obtiene la solución óptima nuestro algoritmo planteado

2. Algoritmo greedy para encontrar el orden óptimo de las batallas

En esta sección se encontrará un análisis completo sobre todas las hipótesis, reglas y consideraciones tomadas por el equipo para la resolución del problema del Señor del Fuego.

Exploramos cómo los principios de los algoritmos greedy se aplican en este contexto, aprovechando la capacidad de tomar decisiones óptimas localmente en cada etapa del problema. Esta estrategia se justifica en la naturaleza del problema, que presenta propiedades de subestructura óptima y de elección óptima, condiciones ideales para la aplicación de un algoritmo greedy.

Resolución de las consignas

1)

En este análisis se va a poner en cuenta como pensamos, planteamos y solucionamos el ejercicio del Trabajo Práctico. Empezamos planteando diferentes ideas y formas, tomando como ejemplo los archivos .TXT dados por la cátedra, planteamos diferentes formas para intentar llegar a los resultados esperados utilizando la sumatoria de

$$\sum T_i * B_i$$

a. utilizar el orden de los archivos .TXT.
b. ordenar por t_i de manera ascendente.
c. ordenar por t_i de manera descendente
d. ordenar por b_i de manera ascendente
e. ordenar por b_i de manera descendente

sin llegar al resultado esperado.

El algoritmo pensado para resolver fue utilizar la misma forma que el problema de la mochila de RPL, en donde plantea que cada elemento tiene un valor y un peso, solo que en el trabajo práctico en vez de ser elementos son batallas y el valor son el tiempo de cada una. Entonces para encontrar un orden correcto lo que hicimos es hacer la división entre

$$t_i \;/\; b_i$$

y ordenar las batallas de manera ascendente con el resultado de esta.

Una vez que está ordenado, calculamos cuánto es la finalización de cada batalla, ya que por el enunciado del trabajo práctico dice "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 $F_j = F_i + t_j$."

Ya teniendo calculada la finalización de cada batalla, pasamos a realizar la sumatoria de estas

$$\sum\ F_i\ *\ B_i$$

quedando como resultado la mínima suma ponderada.

Ejemplo, teniendo las batallas:

$$T_i\ ,\qquad B_i$$

$$23,65$$

$$10,323$$

$$43,923$$

$$76,33$$

Nos quedaría realizando la división

$$T_i\ ,\qquad B_i,\qquad T_I/B_I$$

$$23,65,23/65$$

$$10,323,10/323$$

$$43,923,43/923$$

$$76,33,76/33$$

Y ordenando por T_i/B_i

$$T_i\ ,\ B_i, T_I/B_I$$

10, 323, 10/323

43, 923, 43/923

23, 65, 23/65

76, 33, 76/33

una vez ordenado calculamos los tiempos de finalización de estos.

$F_i\ ,\ B_i, T_I/B_I$

10, 323, 10/323

53, 923, 43/923

76, 65, 23/65

152, 33, 76/33

ya teniendo la finalización de cada batalla, realizamos la sumatoria

$$\sum F_i * B_i$$

quedando como resultado la mínima suma ponderada y el orden de las batallas de la forma

$T_i\ ,\ B_i$

10, 323

43, 923

23, 65

76, 33

2)

Demostración de la optimalidad del algoritmo

Para demostrar que el algoritmo llega siempre a la solución óptima, vamos a realizar primero la demostración por absurdo para 2 batallas, y luego, mediante inducción vamos a demostrar que se cumple para las n batallas.

Demostración para 2 batallas por absurdo

Tenemos 2 batallas:

$$\mathbf{B_1 = (t_1, b_1)}$$

$$\mathbf{B_2 = (t_2, b_2)}$$

Y las vamos a ordenar por coeficiente t_i/b_i de menor a mayor.

Supongamos que este ordenamiento <u>no</u> da siempre la solución óptima. Entonces existe un par de batallas (B_1 B_2) tal que:

$$\frac{t1}{b1} \leq \frac{t2}{b2}$$

(es decir, está ordenado por coeficiente) y que al mismo tiempo se cumple que la suma ponderada de (B_1, B_2) es mayor que la suma ponderada de (B_2, B_1), es decir:

$$t1 \cdot b1 + (t1 + t2) \cdot b2 \;\; \geq \;\; t2 \cdot b2 + (t1 + t2) \cdot b1$$

Si esta desigualdad se cumple para algún (t_1, b_1, t_2, b_2), entonces, el algoritmo no da siempre la solución óptima.

Operando con la desigualdad, y con $b1 \neq 0$ y $b2 \neq 0$ (porque si no, no se podría aplicar el algoritmo, ya que no se puede dividir por cero):

$$\begin{aligned}
t1 \cdot b1 + (t1 + t2) \cdot b2 \;\; &\geq \;\; t2 \cdot b2 + (t1 + t2) \cdot b1 \\
t1 \cdot b1 - (t1 + t2) \cdot b1 \;\; &\geq \;\; t2 \cdot b2 - (t1 + t2) \cdot b2 \\
b1 \cdot (t1 - t1 - t2) \;\; &\geq \;\; b2 \cdot (t2 - t1 - t2) \\
b1 \cdot (t1 - t1 - t2) \;\; &\geq \;\; b2 \cdot (t2 - t1 - t2) \\
-b1 \cdot t2 \;\; &\geq \;\; -b2 \cdot t1 \\
b1 \cdot t2 \;\; &\leq \;\; b2 \cdot t1 \\
\frac{t2}{b2} \;\; &\leq \;\; \frac{t1}{b1}
\end{aligned}$$

$$\frac{t1}{b1} \geq \frac{t2}{b2}$$

Absurdo, porque habíamos dicho al principio que $\frac{t1}{b1} \leq \frac{t2}{b2}$.

Como no pudimos encontrar algún (t_1, b_1, t_2, b_2) que valide la hipótesis, entonces no existe ningún par de batallas (B_1, B_2) cuya suma ponderada resulte peor que la suma ponderada de (B_2, B_1).

Por lo tanto, el algoritmo propuesto siempre encuentra la solución óptima para el caso de 2 batallas.

Demostración para n batallas por inducción

Supongamos que ahora tenemos 3 batallas ordenadas por t_i/b_i. La suma ponderada se calcularía de la siguiente forma:

$$t1 \cdot b1 \ + \ (t1 + t2) \cdot b2 \ + \ (t1 + t2 + t3) \cdot b3$$

Pero a la suma ponderada de las primeras 2 batallas la podríamos escribir como:

$$t1 \cdot b1 \ + \ (t1 + t2) \cdot b2 \ =$$

$$= \ t1 \cdot b1 \cdot \frac{b2}{b2} + \ (t1 + t2) \cdot b2 \cdot \frac{b1}{b1} \ =$$

$$= \ \frac{t1}{b2} \cdot (b1 \cdot b2) \ + \ \frac{t1 + t2}{b1} \cdot (b1 \cdot b2) \ =$$

$$= \ \left(\frac{t1}{b2} + \frac{t1 + t2}{b1}\right) \cdot (b1 \cdot b2) \ =$$

$$= \ t1' \cdot b1'$$

Entonces, la suma ponderada de las 3 batallas la podemos escribir como:

$$t1' \cdot b1' \ + \ (t1 + t2 + t3) \cdot b3$$

Y tenemos de vuelta la suma ponderada entre 2 batallas, caso para el cual ya vimos que el algoritmo siempre encuentra la solución óptima.

Y el mismo procedimiento podríamos aplicar si agregamos una nueva batalla, con lo cual demostramos que algoritmo se cumple para la batalla k+1.

Finalmente, como el algoritmo se cumple para el caso base de 2 batallas (k=2) y luego se cumple para k+1 batallas, queda demostrado que el algoritmo siempre encuentra la solución óptima para todos los casos de batallas posibles.

3)

La complejidad teórica del algoritmo planteado se calculó con la siguiente lógica de análisis:

Partiendo desde la premisa de que solo se usa un archivo .txt que posee todas las batallas con sus pesos, entonces se declara una variable que será complejidad O(1). Luego procede a ingresar a la función leer_batallas

```
def main():
    batallas = []
    leer_batallas(batallas, BATALLAS)

    suma_ponderada, elementos_ordenados = batallas_greedy(batallas)
    print(f"El orden de las batallas que se tienen que hacer estan en el archivo batalla

    escribirResultados(elementos_ordenados, nombre_archivo)
```

En esta función, abre y cierra el archivo txt en una complejidad O(1), gracias a la implementación que posee el lenguaje para el manejo de archivos y luego recorre tantas batallas tenga el archivo, esto se debe a que se recorre cada línea una vez. entonces, siendo n = Número de Batallas -> la complejidad termina en O(n).

Así mismo, se usó la lógica de análisis y concluimos que la complejidad para la escritura los resultados en un .txt sería O(n).

```python
#Leer el archivo txt para almacenar los valores en una lista de lista
def leer_batallas(batallas, nombre_de_archivo):
    #Abrir archivo
    try:
        batallas_archivo = open(nombre_de_archivo, "r")
    except:
        print("Error al abrir el archivo de batallas")
        return

    #Archivo con header "T_i,B_i"
    next(batallas_archivo)

    #Recorrer el archivo y agregar en la lista
    for batalla in batallas_archivo:
        variables = batalla.split(SEPARADOR_BATALLAS)
        batallas.append(variables)

    #Cerrar archivo
    batallas_archivo.close()
```

Luego regresamos al main y procede a llamar a la función “Batallas Greedy”

```python
def main():
    batallas = []
    leer_batallas(batallas, BATALLAS)

    suma_ponderada, elementos_ordenados = batallas_greedy(batallas)
    print(f"El orden de las batallas que se tienen que hacer estan en el archivo batallas.txt

    escribirResultados(elementos_ordenados, nombre_archivo)

#main
```

En ella se encuentra la llamada a dos funciones subyacentes:

```
def batallas_greedy(batallas):
    batallas_ordenadas = ordenar_batallas(batallas)
    suma_ponderada = obtener_suma_ponderada(batallas_ordenadas)
    return suma_ponderada, batallas_ordenadas
```

La primera función ordenar_batallas usa una función sorted() que tiene una complejidad de O(n log n) en el peor de los casos, ya que utiliza un algoritmo de ordenación basado en comparaciones que generalmente tiene una complejidad de O(n log n).

```
#Ordenar la lista segun sea la relacion t_i / b_i
def ordenar_batallas(batallas):
    return sorted(batallas, key=lambda e: int(e[TIEMPO])/int(e[PESO]))
```

La siguiente función a analizar será la función obtener_suma_ponderada:

```
def batallas_greedy(batallas):
    batallas_ordenadas = ordenar_batallas(batallas)
    suma_ponderada = obtener_suma_ponderada(batallas_ordenadas)
    return suma_ponderada, batallas_ordenadas
```

- La función obtener_producto se llama para cada elemento en batallas, lo que resulta en una complejidad de O(n).
- La función reduce aplicada a map también tiene una complejidad de O(n), ya que map recorre cada elemento de batallas y reduce acumula el resultado.

Por lo tanto, la complejidad total de esta parte es O(n).

```python
def obtener_suma_ponderada(batallas):
    tiempo = [0]
    def obtener_producto(batalla):
        tiempo[0] += int(batalla[TIEMPO])
        return int(tiempo[0]) * int(batalla[PESO])
    return reduce(
        lambda b1, b2: b1 + b2,
        map(obtener_producto, batallas),
        0
    )
```

Entonces la función principal, batallas_greedy llama a ordenar_batallas y obtener_suma_ponderada, por lo que su complejidad total es la suma de las complejidades de estas dos funciones. Por lo tanto, la complejidad total de batallas_greedy es O(n log n) + O(n), que simplifica a O(n log n) ya que es el término dominante.

En resumen, la complejidad temporal del algoritmo proporcionado es dominada por la función ordenar_batallas, que tiene una complejidad de **O(n log n)**. Las otras operaciones tienen complejidades menores en comparación.

A su vez, analizamos la variabilidad de los valores T_i y B_i, concluimos de que a pesar de ser nuestra estrategia de orden, en tiempo de ejecución no afecta de forma directamente proporcional. En su lugar, el tiempo de ejecución dependerá netamente de la cantidad de batallas que sean presentadas en archivos .txt

4)

La variabilidad de los valores de T_i , B_i puede afectar la optimalidad del algoritmo de distintas formas. En tiempos de batalla, si el tiempo que se requiere para ganar cada batalla cambia drásticamente, entonces el orden en el que se luchan las batallas puede tener un gran cambio en la suma ponderada de los tiempos de finalización. Batallas que duran más tiempo al principio pueden resultar con una suma ponderada más alta, ya que todas las batallas siguientes se retrasarán. Por lo tanto, puede ser más óptimo luchar primero las batallas más cortas.

En cuanto la importancia de la batalla, Si la importancia de las batallas varía significativamente, entonces el orden en el que se luchan las batallas también puede tener un gran impacto en la suma ponderada de los tiempos de finalización. Batallas menos importantes al principio pueden resultar en una suma ponderada más baja, ya que las batallas más importantes, es decir, las que tienen un mayor peso, se retrasarán. Por lo tanto, puede ser óptimo luchar primero las batallas más importantes.

El algoritmo propuesto tiene en cuenta tanto T_i , B_i al ordenar las batallas por su ratio de importancia a tiempo. Esto significa que el algoritmo dará prioridad a las batallas que son relativamente más importantes y cortas. Sin embargo, si hay mucha variabilidad en T_i , B_i , entonces el orden óptimo de las batallas puede variar significativamente. Por lo tanto, es importante tener en cuenta esta variabilidad al utilizar el algoritmo.

Si los valores de T_i , B_i son negativos, el problema y el algoritmo necesitan ser reevaluados, ya que estos valores negativos podrían no tener un significado físico en el contexto del problema.

Valores negativos para T_i: En el contexto de este problema, un tiempo de batalla negativo no tendría sentido, ya que el tiempo no puede ser negativo. Si se permite un T_i negativo, podría interpretarse como que la batalla ya ha sido ganada antes de comenzar, lo cual es contradictorio.

Valores negativos para B_i**:** Un valor negativo para la importancia de la batalla podría interpretarse como una batalla que el Señor del Fuego preferiría perder en lugar de ganar. Esto podría complicar la optimización, ya que ahora no sólo se está tratando de minimizar el tiempo total, sino también de manejar las preferencias negativas.

En resumen, permitir valores negativos para $T_i\ y\ B_i$, complicaría el problema y requeriría una reevaluación del algoritmo de optimización. Es posible que se necesite un enfoque completamente diferente para resolver el problema en este caso

5)

Usando los ejemplos dados por la catedra, y pasándolos por el algoritmo, podemos obtener los resultados esperados de estos, además utilizando un ejemplo adicional, que calculamos a mano, nos dio el mismo resultado que el algoritmo.

$$T_i, B_i$$

$$10, 50$$

$$37, 42$$

$$167, 21$$

$$566, 3$$

Hacemos la división de

$$T_i,/B_i$$

$$T_i, B_i, T_i/B_i$$

$$10, 50, \frac{10}{50}$$

$$37, 42, \frac{37}{42}$$

$$167, 21, \frac{167}{21}$$

$$566, 3, \frac{566}{3}$$

Sumamos las finalizaciones de cada batalla

$$F_i, B_i$$

$$10, 50$$

$$47, 42$$

$$214, 21$$

$$780, 3$$

Realizamos la sumatoria para obtener la suma ponderada de las batallas

$$\sum_{i=0}^{n} \mathrm{F_i} * B_i$$

Dicha sumatoria nos da como resultado 9308 y el orden de las batallas es

$$T_i, B_i, T_i/B_i$$

$$10, 50, \frac{10}{50}$$

$$37, 42, \frac{37}{42}$$

$$167, 21, \frac{167}{21}$$

$$566, 3, \frac{566}{3}$$

```
El orden de las batallas que se tienen que hacer estan en el archivo batallas.txt y el tiempo es: 9308
```

```
T_i,B_i
10,50
37,42
167,21
566,3
```

6)

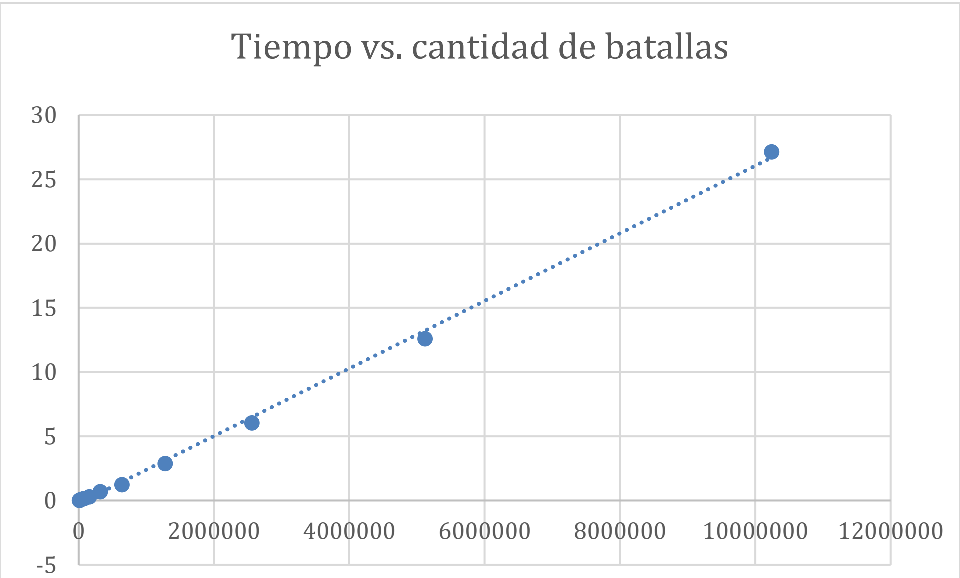
Según los resultados obtenidos (ver apartado *Mediciones*) el algoritmo muestra un comportamiento lineal, lo cual no coincide con la complejidad teórica O(n log n).

Las pruebas de tiempo se hicieron a partir de batallas aleatorias generadas con un algoritmo que se encuentra en el archivo **generator.py**

3. Mediciones

Complejidad del algoritmo

Cantidad de batallas	Tiempo en segundos
10000	0.024569273
20000	0.042161942
40000	0.079659462
80000	0.155718327
160000	0.287978649
320000	0.687191486
640000	1.245909691
1280000	2.873133183
2560000	6.037873268
5120000	12.60144091
10240000	27.15257955



4. Conclusiones

En conclusión, este informe ha presentado un enfoque basado en algoritmos greedy para optimizar la estrategia militar del Señor del Fuego, demostrando su aplicabilidad y eficacia en la resolución de problemas complejos. A través de un análisis detallado, se ha demostrado que el algoritmo propuesto, que se basa en la toma de decisiones óptimas localmente en cada etapa del problema, es efectivo para maximizar el impacto de las victorias en la Nación del Fuego.

Se ha explorado la complejidad teórica del algoritmo, revelando que su eficiencia está dominada por la función de ordenamiento de batallas, con una complejidad de O(n log n). Además, se ha analizado la variabilidad de los valores de tiempo de batalla (T_i) y su importancia (B_i), demostrando que, aunque pueden afectar la optimalidad del algoritmo, no lo hacen de manera directamente proporcional en tiempo de ejecución. En cambio, el tiempo de ejecución depende principalmente de la cantidad de batallas presentadas.

El algoritmo propuesto ha demostrado ser una herramienta valiosa para el Señor del Fuego, permitiéndole tomar decisiones óptimas en cada etapa del conflicto. Sin embargo, es importante tener en cuenta que la variabilidad en los valores de T_i y B_i puede afectar la efectividad del algoritmo, lo que subraya la necesidad de una evaluación continua y ajustes según las circunstancias específicas.

La adaptabilidad y eficiencia de los algoritmos greedy los convierten en herramientas valiosas para la toma de decisiones en entornos dinámicos y complejos, como los que enfrenta el Señor del Fuego.