Teoría de algoritmos

(75.29) Curso Buchwald - Genender

Trabajo Práctico 1: Algoritmos Greedy en la nación del fuego

Fecha de entrega: 8 de abril 2024

Integrantes:

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

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

Algoritmo Greedy

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

1. utilizar el orden de los archivos .TXT.
2. ordenar por de manera ascendente.
3. ordenar por de manera descendente
4. ordenar por de manera ascendente
5. ordenar por 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

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 _Fj_=_tj_, en cambio si la batalla *j* se realiza justo después de la batalla *i*, entonces _Fj_=_Fi_+_tj_.”

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

quedando como resultado la mínima suma ponderada.

Ejemplo, teniendo las batallas:

23, 65

10, 323

43, 923

76, 33

Nos quedaría realizando la división

23, 65,

10, 323,

43, 923,

76, 33,

Y ordenando por

10, 323,

43, 923,

23, 65,

76, 33,

una vez ordenado calculamos los tiempos de finalización de estos.

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

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

10, 323

43, 923

23, 65

76, 33

2)

Prueba directa

2) LO DEMOSTRAMOS DE FORMA DIRECTA

USEMOS UN ARREGLO DE 2 BATALLAS QUE ASUMIMOS YA ORDENADO POR $\frac{t}{b}$ (MENOR)

ORDENADO $[(t_1, b_1), (t_2, b_2)]$ { DESORDENADO $[(t_2, b_2), (t_1, b_1)]$

→ SE CUMPLE QUE $\frac{t_1}{b_1} \leq \frac{t_2}{b_2}$

→ PODEMOS DECIR QUE $\left.\begin{matrix} t_2 = \alpha t_1 \\ b_2 = \beta b_1 \end{matrix}\right\} \frac{t_2}{b_2} \geq \frac{t_1}{b_1} \Leftrightarrow \frac{\alpha t_1}{\beta b_1} \geq \frac{t_1}{b_1} \Leftrightarrow$

$\Leftrightarrow \frac{\alpha}{\beta} \geq 1 \Leftrightarrow \alpha \geq \beta$ (1)

$\beta > 0 \wedge \alpha > 0$

CALCULAMOS LA SUMA PONDERADA ORDENADO:

$\Sigma_1 = t_1 b_1 + (t_1 + t_2) b_2$

$= t_1 b_1 + (t_1 + \alpha t_1) \beta b_1$

$= t_1 b_1 + t_1 \beta b_1 + \alpha t_1 \beta b_1$

$= t_1 b_1 + \beta t_1 b_1 + \alpha\beta t_1 b_1 = t_1 b_1 (1 + \alpha\beta + \beta)$

CALCULAMOS LA SUMA PONDERADA DESORDENADO

$\Sigma_2 = t_2 b_2 + (t_2 + t_1) b_1$

$= \alpha t_1 \beta b_1 + (\alpha t_1 + t_1) b_1$

$= \alpha t_1 \beta b_1 + \alpha t_1 b_1 + t_1 b_1$

$= t_1 b_1 \alpha\beta + \alpha t_1 b_1 + t_1 b_1 = t_1 b_1 (1 + \alpha\beta + \alpha)$

COMPARO AMBOS

¿ $\Sigma_1 \leq \Sigma_2$?

$t_1 b_1 (1 + \alpha\beta + \beta) \leq t_1 b_1 (1 + \alpha\beta + \alpha)$

$\beta \leq \alpha$

TRUE, POR (1) ✓

Σ_1 SIEMPRE ES $\leq$ A Σ_2, es decir, "la suma ponderada luego de ordenar por menor $\frac{t}{b}$" es siempre $\leq$ a "la suma ponderada antes de ordenar por menor $\frac{t}{b}$" para el caso de 2 batallas

Si aplicamos esta regla de forma acumulativa ("greedy"), ocurre que:

B_1 con B_2 → SUMA MÍNIMA ORDENADO

B_2 con B_3 → SUMA MÍNIMA ORDENADO

B_3 con B_4 → SUMA MÍNIMA ORDENADO

B_4 con B_5 → SUMA MÍNIMA ORDENADO

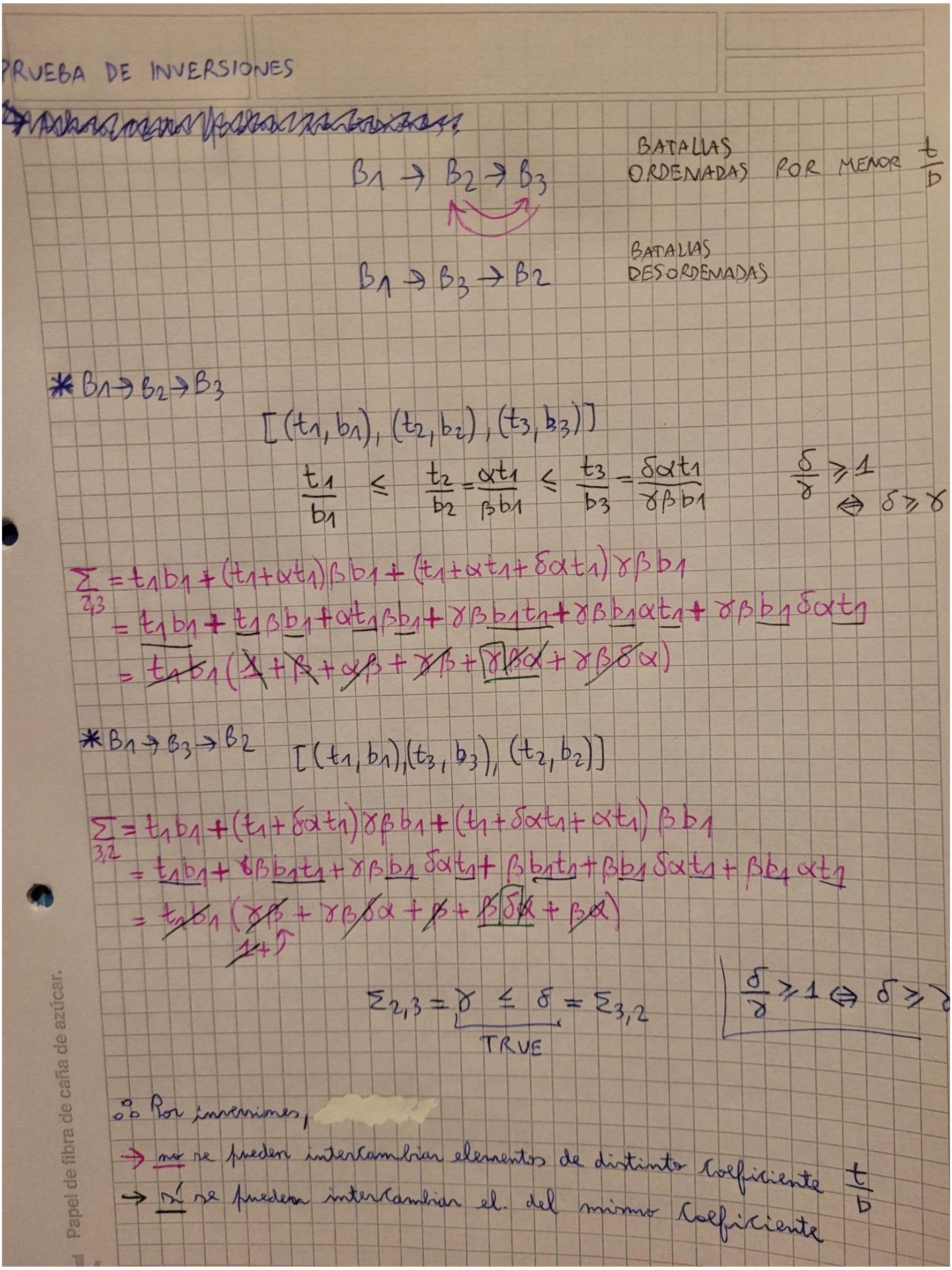
...

B_{n-1} con B_n → SUMA MÍNIMA ORDENADO

B = batalla

y esto sumado a los resultados de la PRUEBA DE INVERSIONES, nos demuestran que la solución es óptima ya que en cada paso nos aseguramos de obtener la suma mínima y demostramos que los elementos deben estar ordenados sí o sí de menor a mayor coeficiente para garantizar que el ordenamiento siempre de una sumatoria igual o más pequeña a la anterior como resultado.

Prueba por inversiones



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).

```
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)

    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”

```
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)
```

En ella se encuentra la llamada a dos funciones subyacentes:

```python
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).

```python
#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”:

```python
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.

```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
    )
```

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

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 Ti y Bi, 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 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 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 , 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 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 : 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 negativo, podría interpretarse como que la batalla ya ha sido ganada antes de comenzar, lo cual es contradictorio.

Valores negativos para : 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 , 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)

6)

Según los resultados obtenidos (ver apartado *Mediciones*) el algoritmo parece comportarse de manera lineal, lo cual no coincide con nuestras predicciones. De alguna forma es mejor, lo cual no tiene sentido, porque el ordenamiento tarda O(n log n), y el resultado global es O(n).

Las pruebas se pueden repetir utilizando el código de nuestro repositorio:

https://github.com/MatiVazquezMorales/TDA-tp1/tree/batch_test

En la rama **batch_test** hay 3 archivos:

· **tp1.py:** al ejecutar este script, se procesan los archivos de batallas guardados en la carpeta "generated" (si fueron sets de datos generados usando el archivo generator.py) o en la carpeta "tests\data" (los archivos de prueba de la cátedra) si se cambia el valor de la variable DATASET_PATH para que apunte a esa carpeta.

· **tp1_tests.py:** al ejecutar este script, se corren los tests unitarios con los archivos de batallas de la cátedra. Todos los tests pasan con nuestro algoritmo, tal como se espera.

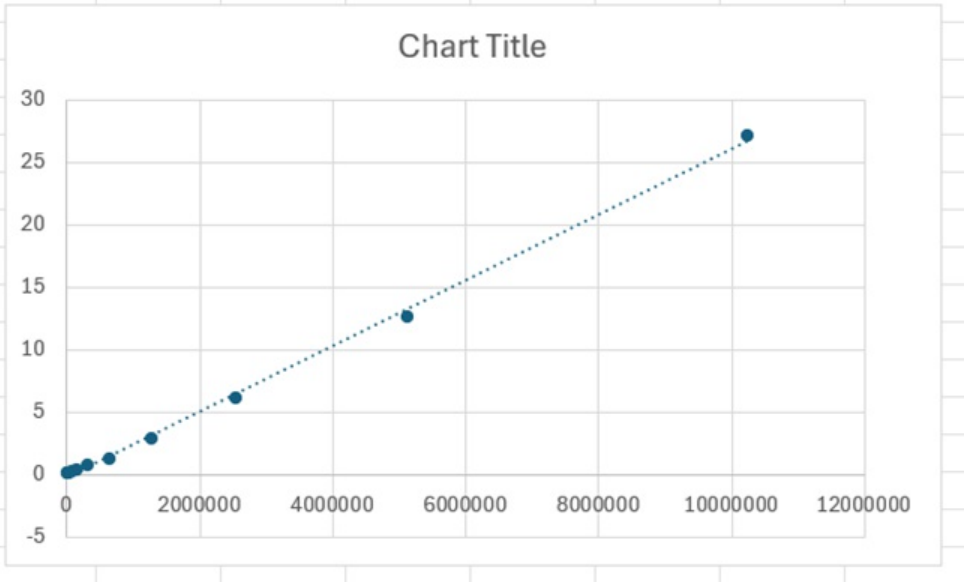
· **generator.py:** al ejecutar este script, se generan casos de prueba aleatorios de batallas con distintos tamaños de datos (los tamaños a generar están indicados en la variable "amounts"). Todos los archivos creados se guardan en la carpeta "generated". Para procesar estos archivos generados hay que ejecutar tp1.py

Al ejecutar tp1.py, además de guardar las batallas ordenadas en sus respectivos archivos nuevos (por ejemplo: "100.txt" ---> "100_solved.txt"), se imprime por consola la suma ponderada calculada para cada caso ejecutado, y al final se imprime un "csv" con tuplas **(batallas_tamaño, tiempo_en_segundos)** donde se observa la complejidad medida del algoritmo.

Mediciones

Complejidad del algoritmo

batallas_amount	time_in_seconds
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



Conclusión

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 (Ti) y su importancia (Bi), 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 Ti y Bi 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.