

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)

# **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 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

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)**

**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:

**B1 = (t1, b1)**

**B2 = (t2, b2)**

Y las vamos a ordenar por coeficiente ti/bi de menor a mayor.

Supongamos que este ordenamiento no da siempre la solución óptima. Entonces existe un par de batallas (B1 B2) tal que:

(es decir, está ordenado por coeficiente) y que al mismo tiempo se cumple que la suma ponderada de (B1, B2) es mayor que la suma ponderada de (B2, B1), es decir:

Si esta desigualdad se cumple para algún (t1, b1, t2, b2), entonces, el algoritmo no da siempre la solución óptima.

Operando con la desigualdad, y con y (porque si no, no se podría aplicar el algoritmo, ya que no se puede dividir por cero):

**Absurdo**, porque habíamos dicho al principio que  **.**

Como no pudimos encontrar algún (t1, b1, t2, b2) que valide la hipótesis, entonces no existe ningún par de batallas (B1, B2) cuya suma ponderada resulte peor que la suma ponderada de (B2, B1).

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 ti/bi. La suma ponderada se calcularía de la siguiente forma:

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

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

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:

Pantalla de computadora con letras

Descripción generada automáticamente con confianza media

la línea if len(sys.argv) != 2: verifica si la longitud de la lista sys.argv es diferente de 2, si la longitud no es igual a 2, se imprime un mensaje de ejemplo y devuelve, la complejidad de esta verificación es constante O(1) ya que no depende del tamaño de los datos de entrada.

La otra linea if not sys.argv[1].split(".")[0].isnumeric(): verifica si el primer argumento no es numérico, la complejidad de esta verificación depende de la longitud del primer argumento, siendo esta O(n).

Texto

Descripción generada automáticamente

La lectura de archivos y manejo de rutas, tiene una complejidad constante O(1).

En la iteración sobre los archivos se hace el llamado a la función resolver\_probelma\_batallas que va a ser analizada más adelante.

En la impresión de resultados, tiene una complejidad lineal O(n) debido a la iteración sobre las batallas

Por lo tanto sin analizar resolver\_problema\_batalla, tp1\_solver tiene complejidad O(n).

Texto

Descripción generada automáticamente

La función lee el archivo y llama a tp1\_batallas\_solver, la lectura de este depende del numero de líneas del archivo, por lo tanto es O(n).

también hace el llamado a otras 2 funciones, tp1\_batallas\_solver y escribir\_resultados.

Texto

Descripción generada automáticamente

La función hace el llamado a otras funciones.

Texto

Descripción generada automáticamente

La creación del nombre del archivo y la apertura tiene una complejidad constante O(1), y la iteración sobre las batallas tiene una complejidad lineal O(n) donde n es el número de batallas.

Texto

Descripción generada automáticamente

Hace el llamado a la funcion ordenar\_batallas y a obtener\_suma\_ponderada y retorna la suma y el orden de las batallas.

Una captura de pantalla de un celular

Descripción generada automáticamente con confianza media

Implementa el ordenamiento de batallas según T\_i/B\_i, utiliza la función sorted que tiene un coste de O(n log n) y la función lambda que tiene complejidad constante.

Texto

Descripción generada automáticamente

La función map y la función reduce recorren la lista de batallas una vez, por lo tanto, la complejidad es lineal O(n) donde n es el número de batallas.

Texto

Descripción generada automáticamente

La lectura de archivo tiene una complejidad lineal O(n), donde n es el número de líneas del archivo.

En conclusión, la complejidad algorítmica de este algoritmo es O(n log n) por el ordenamiento utilizando la función sorted, ya que el resto de las operaciones son lineales o en mejor caso constantes.

**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)**

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.

10, 50

37, 42

167, 21

566, 3

Hacemos la división de

10, 50,

37, 42,

167, 21,

566, 3,

Sumamos las finalizaciones de cada batalla

10, 50

47, 42

214, 21

780, 3

Realizamos la sumatoria para obtener la suma ponderada de las batallas

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

10, 50,

37, 42,

167, 21,

566, 3,



Texto

Descripción generada automáticamente

**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**

# **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 |

# 

# **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 (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.