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 , B_i

23, 65

10, 323

43, 923

76, 33

Nos quedaría realizando la división

T_i , B_i, 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 no 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:

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

$$= \;\; t1 \cdot b1 \; + \; (t1 + t2) \cdot \alpha \cdot b1, \;\; \alpha \in \mathbb{R} =$$

$$= \;\; [\, t1 + \alpha \cdot (t1 + t2) \,] \cdot b1 \;\; =$$

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

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:

```python
def main():
    if len(sys.argv) != 2:
        print("Ejemplo de uso: python3 tp1.py 50000.txt")
        return

    if not sys.argv[1].split(".")[0].isnumeric():
        print("Ejemplo de uso: python3 tp1.py 50000.txt")
        return

    path = sys.argv[1]

    tp1_solver(path, is_filename_only = True)
```

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).

```python
def tp1_solver(path, is_filename_only):
    try:
        if is_filename_only:
            archivos_batallas = [ path ]
            path_ = os.getcwd()
        else:
            archivos_batallas = sorted( \
                os.listdir(path), \
                key = lambda x: int(x.split(".txt")[0]) \
            )
            path_ = path
    except:
        print("Could not find dataset folder.")
        return

    batallas = []

    for nombre_archivo in archivos_batallas:
        suma_ponderada, tiempo = resolver_problema_batallas(f"{path_}\\{nombre_archivo}")
        batallas.append((nombre_archivo.split(".")[0], tiempo))

        nombre_archivo_ = nombre_archivo.split(".")[0] \
            + "_solved." \
            + nombre_archivo.split(".")[1]

        print(f"El orden de las batallas resuelto está \
              en el archivo {nombre_archivo_} y la suma ponderada \
              de los tiempos de finalización es: {suma_ponderada}")

    print("batallas,time_in_seconds")
    for b in sorted(batallas, key=lambda x: int(x[0])):
        print(f"{b[0]},{b[1]}")
```

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).

```python
def resolver_problema_batallas(data_filepath):
    start_time = time.time()
    suma_ponderada, batallas_ordenadas = tp1_batallas_solver(data_filepath)
    end_time = time.time()

    escribir_resultados(data_filepath, batallas_ordenadas)

    return suma_ponderada, (end_time - start_time)
```

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.

```python
def tp1_batallas_solver(batallas_path):
    batallas = []
    leer_batallas(batallas, batallas_path)
    suma_ponderada, batallas_ordenadas = batallas_greedy(batallas)
    return suma_ponderada, batallas_ordenadas
```

La función hace el llamado a otras funciones.

```python
def escribir_resultados(batallas_path, batallas_ordenadas):
    solved_path = "".join(batallas_path.split(".")[0:-1]) \
        + "_solved." \
        + batallas_path.split(".")[-1]

    archivo_resultados = open(f"{solved_path}", "w")
    archivo_resultados.write(HEADER_BATALLAS)

    for i in range(len(batallas_ordenadas)):
        batalla = batallas_ordenadas[i]
        to_write = f"\n{batalla[0].replace("\n", "")},{batalla[1].replace("\n", "")}"
        archivo_resultados.write(to_write)
```

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.

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

Hace el llamado a la funcion ordenar_batallas y a obtener_suma_ponderada y retorna la suma y el orden de las batallas.

```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]))
```

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.

```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
    )
```

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.

```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()
```

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 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).

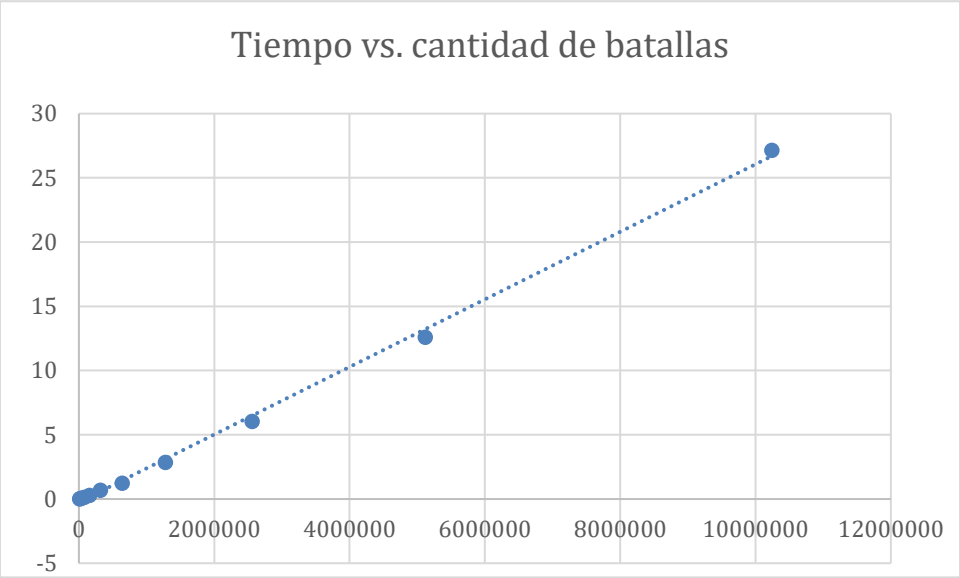
Las pruebas de tiempo se hicieron a partir de batallas aleatorias generadas con un algoritmo que se encuentra en el archivo **generator.py**

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.