.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 1
Algoritmos Greedy en la Nación del Fuego

30 de abril de 2024

Ana Gabriela Gutson	**Lucas Franciulli**	**Mariana Juarez Goldemberg**
105853	107059	108441

1. Introducción

Es el año 10 AG, y somos asesores del Señor del Fuego (líder supremo de la Nación del Fuego). El Señor del Fuego cuenta con un ejército de Maestros Fuego, muy temidos en el mundo. Tiene varias batallas con las cuales lidiar: una contra el Templo Aire del Este, otra en la Tribu del Agua del Norte, otra en la Isla de Kyoshi, una muy importante en Ba Sing Se (capital del Reino de la Tierra), y muchas otras más. 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.

1.1. Consigna

1. Hacer un análisis del problema, y 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 cual se minimiza $\sum_{i=1}^{n} b_i F_i$.
2. Demostrar que el algoritmo planteado obtiene siempre la solución óptima.
3. Escribir el algoritmo planteado. Describir y justificar la complejidad de dicho algoritmo. Analizar si (y cómo) afecta la variabilidad de los valores de t_i y b_i a los tiempos del algoritmo planteado.
4. Analizar si (y cómo) afecta la variabilidad de los valores de t_i y b_i a la optimalidad del algoritmo planteado.
5. Realizar ejemplos de ejecución para encontrar soluciones y corroborar lo encontrado.
6. Hacer mediciones de tiempos para corroborar la complejidad teórica indicada. Agregar los casos de prueba necesarios para dicha corroboración. Realizar gráficos correspondientes.

1.2. Ejemplo

Dadas 10 batallas, cada una con el T_i y B_i que figuran en el Cuadro 1, la mínima suma ponderada de los tiempos de finalización es de 8858347.

1.3. ¿Qué es un algoritmo Greedy?

Son algoritmos donde aplicamos una regla sencilla de manera iterativa para obtener en cada iteración el óptimo local. De esta manera, a partir de obtener la serie de óptimos locales, obtenemos el óptimo global.

Ti	Bi
595	889
930	860
585	320
276	546
356	61
97	757
813	613
797	947

Cuadro 1: Ejemplo de batallas

Resolución

2. Análisis del problema

El problema a resolver radica en la optimización de la suma ponderada de los tiempos de finalización de las batallas, con el objetivo de minimizar el valor resultante:

$$\sum_{i=1}^{n} b_i F_i$$

Donde:

- F_i: Felicidad obtenida en el momento donde termina la batalla i.
- b_i: Importancia de la batalla i.

Contamos con el tiempo que necesita el ejército de Maestros Fuego para ganar la batalla i (t_i). Por lo tanto se define a la felicidad F_j como $F_j = F_i$ si la batalla j es la primera realizada o $F_j = F_i + t_j$ si la batalla j se finaliza luego de la batalla i.

Además, tendremos en cuenta que al Señor del Fuego no le da lo mismo el orden en el que se realizan las batallas, porque comunicar la victoria a su nación en diferentes batallas genera menos impacto si pasa mucho tiempo.

Debemos diseñar un algoritmo **Greedy** que nos permita obtener el orden de realización de las batallas tal que la suma poderada de los tiempos de las mismas sea mínimo.

3. Análisis de posibles soluciones

Al comienzo del diseño del algoritmo, en busca del que nos de la solución óptima, iteramos sobre diferentes soluciones hasta encontrar la deseada.

Nuestro primer enfoque se centró en el Greedy intuitivo y fácil de implementar, donde definimos ordenar las batallas según las reglas que veremos más abajo.

Para ello, desarrollaremos sobre un set de datos de la forma:

$$[(t_1,b_1),(t_2,b_2),(t_3,b_3)]$$

3.1. Importancia (peso) en forma ascendente

La primera regla Greedy que ha surgido fue ordenar las batallas según su peso, de forma ascendente e ir obteniendo el elemento con el peso más chico. Este ordenamiento funcionaría en ciertos casos ya que damos prioridad a las batallas más importantes.

3.1.1. Contraejemplo

Ahora veremos un contraejemplo en el cual este algoritmo Greedy no es óptimo:

En el caso donde todas las batallas tengan la misma importancia, no se logrará la solución óptima, ya que podría ordenarse de cualquier forma sin tener en cuenta el tiempo.

A continuación veremos este caso:

$$[(t_1,b_1),(t_2,b_2),(t_3,b_3)] = [(3,2),(2,2),(4,2)]$$

Array después de ordenar por peso:

$$[(3,2),(2,2),(4,2)]$$

Ahora simplemente tenemos que tomar el primer elemento, ya que es el que tendría el menor peso, e iniciaria los calculos para la suma ponderada. Es decir:

$$Suma\ Ponderada : (3*2) + ((3+2)*2) + ((3+2+4)*2) = 34$$

Cuando el resultado óptimo de la suma ponderada debería ser: 32.

Siendo el orden correcto:

$$Suma\ Ponderada : (2*2) + ((2+3)*2) + ((2+3+4)*2) = 32$$

3.2. Tiempo de batalla en forma ascendente

La segunda regla Greedy que ha surgido fue ordenar por el tiempo de finalización en vez del peso. Tener este ordenamiento tiene sentido ya que abordaríamos primero las batallas que terminan rápidamente.

3.2.1. Contraejemplo

Ahora veremos el contraejemplo de este caso:

Ordenando por el tiempo, en el caso de donde todas las batallas tengan la misma importancia, no se logrará la solución óptima, ya que podría ordenarse de cualquier forma sin tener en cuenta el peso.

A continuación veremos este caso:

$$[(t_1,b_1),(t_2,b_2),(t_3,b_3)] = [(2,3),(2,2),(2,4)]$$

Array después de ordenar por tiempo:

$$[(2,3),(2,2),(2,4)]$$

Ahora simplemente tenemos que tomar el primer elemento de este array ya que tendría el menor tiempo de finalización e iniciamos los calculos para la suma ponderada. Es decir:

$$Suma\ Ponderada : (2*3) + ((2+2)*2) + ((2+2+2)*4) = 38$$

Cuando el resultado óptimo de la suma ponderada debería ser: 32

Siendo el orden correcto:

$$Suma\ Ponderada : (2*4) + ((2+2)*3) + ((2+2+2)*2) = 32$$

3.3. Coeficiente de la importancia de la batalla y el tiempo en forma descendente

La última regla Greedy que ha surgido fue ordenar por el coeficiente entre la importancia de la batalla (b_i) y el tiempo que ésta conlleva (t_i), de forma descendente. Este ordenamiento tiene en cuenta ambas variables, por lo cual desarrollamos la solución con esta regla.

4. Algoritmo para obtener el óptimo

A continuación, desarrollaremos el análisis del algoritmo que nos ha dado el resultado óptimo.

4.1. ¿El algoritmo elegido es Greedy?

Definimos como heurística a la hora de ordenar el orden descendente del coeficiente generado por la división $\frac{b_i}{t_i}$ de cada batalla i. Esto cumple con lo que llamamos regla Greedy, la cual se irá repitiendo iterativamente para conseguir los óptimos locales hasta llegar al óptimo global. Así, definimos que nuestra propuesta es un algoritmo Greedy.

4.2. Implementación en código funcional del algoritmo

Para organizar nuestro conjunto de datos de batallas, que consiste en pesos b_i y tiempos t_i, comenzamos utilizando el algoritmo MergeSort.

Después de ordenar los datos, iteramos sobre ellos para calcular la felicidad que cada batalla aporta, acumulando este valor en la variable $felicidad\_actual$, y calculamos cómo influye en la suma ponderada de los tiempos de finalización, incrementando $sumatoria\_total$ en consecuencia.

De esta manera, determinamos el orden de las batallas y calculamos la suma ponderada de los tiempos de finalización.

```
def merge_sort(arr):
    if len(arr) > 1:
        mitad = len(arr) // 2
        mitad_izq = arr[:mitad]
        mitad_der = arr[mitad:]

        merge_sort(mitad_izq)
        merge_sort(mitad_der)

        merge(arr, mitad_izq, mitad_der)

def merge(arr, izq, der):
    i = j = k = 0

    while i < len(izq) and j < len(der):
        if izq[i][1] / izq[i][0] > der[j][1] / der[j][0]:
            arr[k] = izq[i]
            i += 1
        else:
            arr[k] = der[j]
            j += 1
        k += 1

    while i < len(izq):
        arr[k] = izq[i]
        i += 1
        k += 1

    while j < len(der):
        arr[k] = der[j]
        j += 1
        k += 1
```

```
33
34 def calcular_orden_optimo(datos_batallas):
35     merge_sort(datos_batallas) # O(n*log(n))
36
37     orden_de_batallas = []
38
39     sumatoria_total = 0
40     felicidad_actual = 0
41     for i in range(len(datos_batallas)): #O(n)
42         tiempo, peso = datos_batallas[i]
43         sumatoria_total += (felicidad_actual + tiempo) * (peso)
44         felicidad_actual += tiempo
45         orden_de_batallas.append((tiempo, peso))
46
47     #O(n*log(n) + #O(n) = #O(n*log(n))
48     return sumatoria_total, orden_de_batallas
```

4.3. Complejidad

Al código mostrado anteriormente, le agregamos las complejidades de los procesos relevantes, el resto, se trata de procesos con complejidad constante: $O(1)$.

1. Merge Sort

```
1     merge_sort(datos_batallas) #O(n*log(n))
```

La complejidad temporal del MergeSort se describe mediante el Teorema Maestro.

El Teorema Maestro establece que si un algoritmo divide un problema en a subproblemas, cada uno de los cuales es de tamaño n/b, y resuelve cada subproblema recursivamente con un costo de $T(n/b)$, mientras que el costo de combinar las soluciones recursivas es $D(n)$, entonces la complejidad total del algoritmo está dada por:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + D(n)$$

Para resolver la ecuación de recurrencia y determinar la complejidad de Merge Sort, aplicamos el Teorema Maestro.

En este caso, tenemos $a = 2$ (porque dividimos el problema en dos subproblemas), $b = 2$ (porque cada subproblema tiene la mitad del tamaño del problema original), y $D(n) = O(n)$ (el tiempo necesario para fusionar las dos mitades ordenadas), resultando en la ecuación de recurrencia:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$

Para resolver la ecuación de recurrencia y determinar la complejidad de Merge Sort, aplicamos el Teorema Maestro. Primero identificamos el caso del Teorema Maestro. Calculamos $\log_b a$:

$$\log_2 2 = 1$$

Como $\log_b a = 1$, estamos en el caso 2 del Teorema Maestro.

Para el caso 2, el teorema maestro nos dice que la complejidad del algoritmo es $O(n^{\log_b a} \log n)$.

Por lo tanto, la complejidad es $O(n \log n)$.

2. Iteración sobre datos de batalla

```
1     for i in range(len(datos_batallas)): #O(n)
2         tiempo, peso = datos_batallas[i]
3         sumatoria_total += (felicidad_actual + tiempo) * (peso)
4         felicidad_actual += tiempo
5         orden_de_batallas.append((tiempo, peso))
```

En el bucle donde recorremos todas las n batallas, la complejidad es $O(n)$, y cada operación dentro del bucle, como acceder a los elementos de los datos de la batalla y realizar cálculos simples, tiene una complejidad constante $O(1)$. Por lo tanto, la complejidad total en este bloque es $O(n)$.

Finalmente, el desarrollo final de la complejidad es:

$$\mathcal{T}(n) = \mathcal{O}(n * log(n)) + \mathcal{O}(n) = \mathcal{O}(n * log(n))$$

4.4. ¿Es realmente el óptimo?

Este ordenamiento que hemos elegido para calcular la mínima suma ponderada (ordenando por el coeficiente $\frac{b_i}{t_i}$ de forma descendente), parecería ser el óptimo.

Para probarlo, utilizaremos inversiones. Llamaremos al schedule óptimo posible obtenido por nuestro algoritmo Greedy S y a otro schedule alternativo K, el cual debe tener mínimamente una inversión.

Definimos una inversión como un par de batallas i, j, donde en el schedule K, la batalla i viene antes que la batalla j. Mientras que, para nuestro schedule "óptimo" S, la batalla j viene antes que la batalla i, pues $\frac{bj}{tj} \geq \frac{bi}{ti}$.

Si logramos demostrar que intercambiando las batallas i, j invertidas no incrementamos la suma ponderada podemos intercambiar hasta no tener más inversiones. Con esto llegaremos a nuestra solución Greedy propuesta con el schedule S. De esta manera podemos afirmar que la suma ponderada del schedule S no es mayor que la suma ponderada para otro schedule K. Afirmando que es óptimo.

Llamemos al tiempo de finalización de las batallas antes de las batallas i y j como Z. Antes del intercambio, la contribución a la suma ponderada de las batallas i, j (recordemos, en el K, i viene antes que j) será:

$$b_i(Z + t_i) + b_j(Z + t_i + t_j)$$

Y luego del intercambio:

$$b_j(Z + t_j) + b_i(Z + t_j + t_i)$$

Si calculamos la diferencia entre antes del intercambio y después tenemos:

$$\begin{aligned} &bi(Z + t_i) + bj(Z + t_i + t_j) - [bj(Z + t_j) + bi(Z + t_j + t_i)] = \\ &biZ + bi \cdot ti + bjZ + bj \cdot ti + bj \cdot tj - bjZ - bj \cdot tj - biZ - bi \cdot tj - bi \cdot ti = \\ &bj \cdot ti - bi \cdot tj \end{aligned}$$

Sabiendo que $\frac{bj}{tj} \geq \frac{bi}{ti}$, la diferencia está acotada por sobre el 0, por lo tanto la suma ponderada de los tiempos de finalización no aumentará dado el intercambio.

4.5. Mediciones de tiempo y complejidad algoritmica

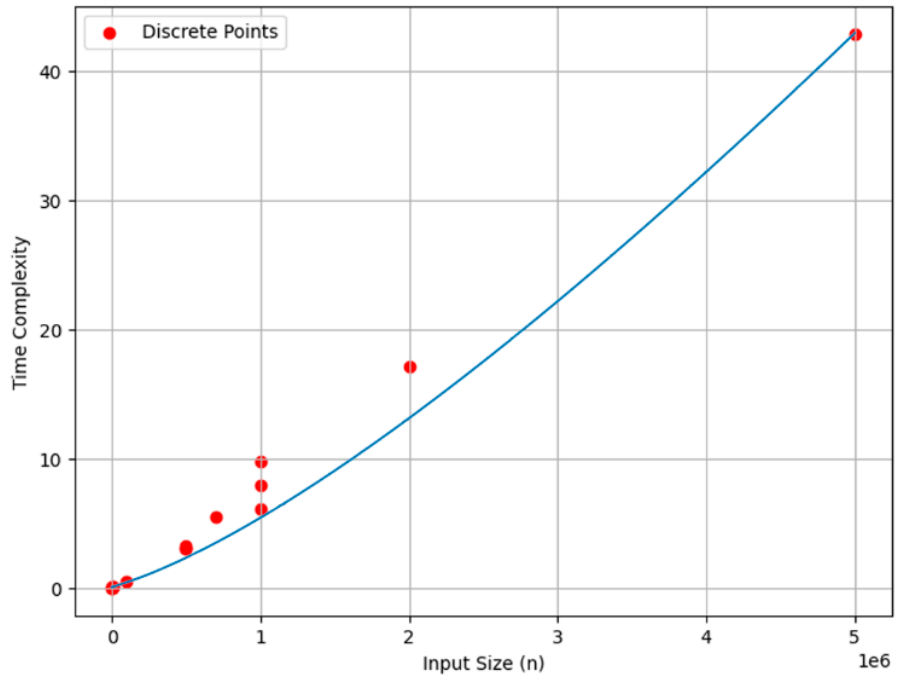
A continuación graficaremos todos los ejemplos que se encuentran en la carpeta Ejemplos de nuestro repositorio y observaremos si la tendencia de los puntos discretos se asemeja a la Big O Notation que declaramos con anterioridad.

Resultados:

Archivo: (cantidad de elementos, tiempo de ejecución)

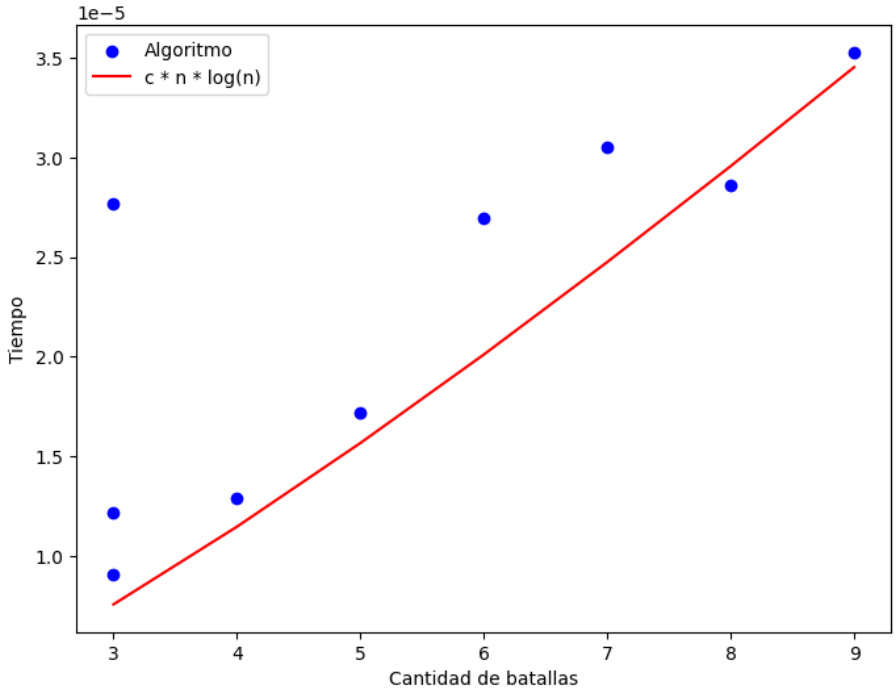
10.txt: (10, 0.0)
50.txt: (50, 0.1)
100.txt: (100, 0.0)
1000.txt: (1000, 0.0)
5000.txt: (5000, 0.1)
10000.txt: (10000, 0.1)
100000.txt: (100000, 0.5)
500000.txt: (500000, 3.2)
TiBajoBiAltoMedioMillon.txt: (500000, 3.0)
TiAltoBiBajoMedioMillon.txt: (500000, 3.0)
700000.txt:(700000, 5.5)
1000000.txt: (1000000, 9.8)
TiAltoBiBajo1millon.txt: (1000000, 6.1)
TiBajoBiAlto1millon.txt: (1000000, 7.9)
2000000.txt: (2000000, 17.1)
5000000.txt:(5000000, 42.9)

Gráfico

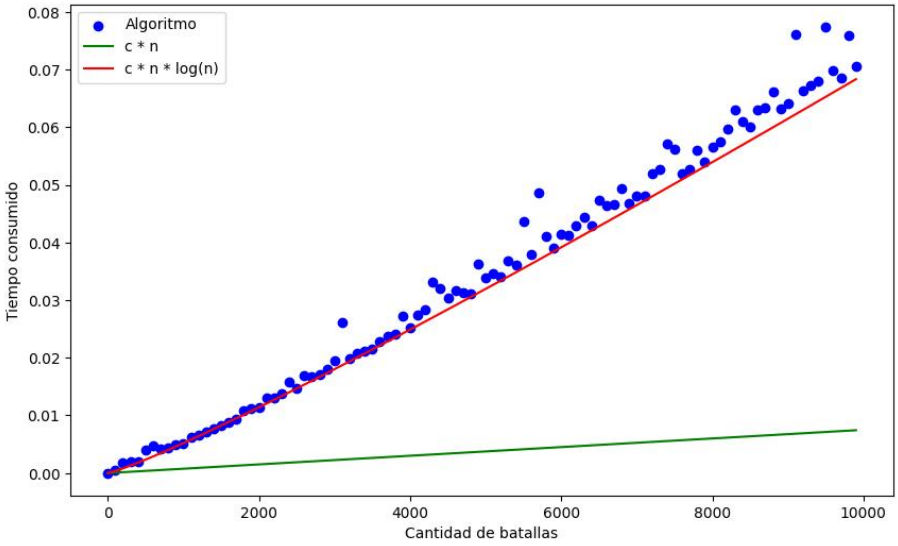


A partir de unas pruebas generadas, además de las otorgadas por la cátedra, se verificó que nuestro algoritmo nos otorgue la solución óptima.

Graficamos con variabilidad de los datos b_i y t_i con respecto al tiempo de ejecución. Observamos que el tiempo de ejecución no se ve afectado por la misma. Además, observamos que sigue la complejidad que resultó de nuestro análisis.



Ahora, si aumentamos la cantidad de muestras (arreglos de diferentes largos, cuyos elementos fueron generados por la librería *random* de Python), observamos que la complejidad también se mantiene. Tuvimos ciertos casos donde el tiempo de ejecución aumentaba considerablemente dado el tamaño de los conjuntos. Utilizamos la librería *time* para medir el tiempo que tardaba el algoritmo en calcular la solución para cada arreglo.

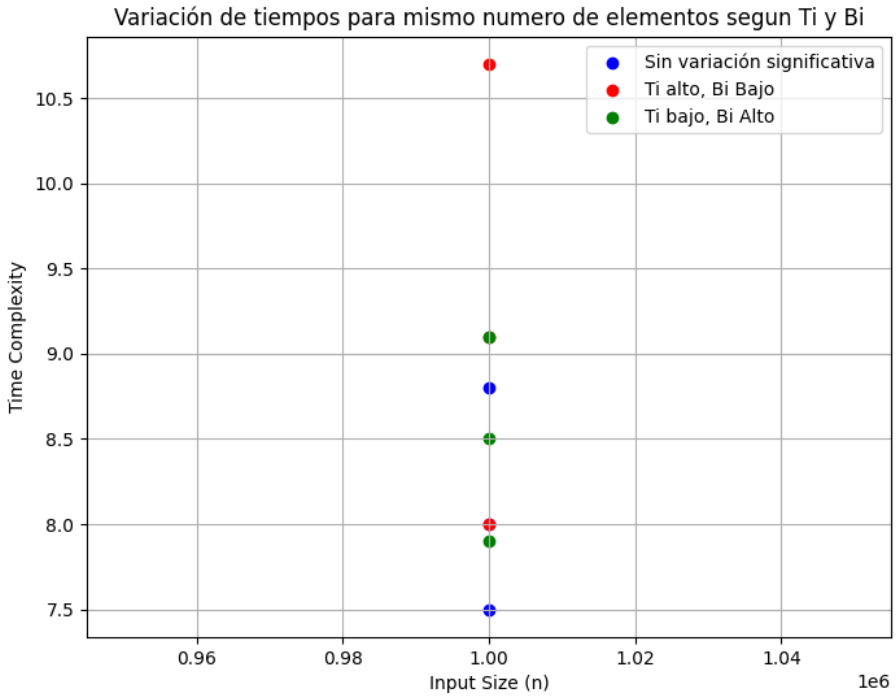


Como se puede observar, los puntos tienden a seguir la complejidad algoritmica de

$$\mathcal{T}(n) = \mathcal{O}(n * log(n))$$

Cabe destacar que el tiempo de ejecución con los archivos de "1000000.txt", "TiAltoBiBajo1millon.txtz "TiBajoBiAlto1millon.txt"fueron realizados reiteradas veces y pudimos determinar que realmente no tiene un impacto la variación de estos.

Esto se puede apreciar en el siguiente gráfico donde utilizamos 3 variaciones de los archivos previamente mencionados.



5. Conclusión

Al finalizar el trabajo práctico, hemos llegado a concluir que, resolver este problema utilizando algoritmos Greedy resultó bastante eficiente a la hora de encontrar óptimos locales y el buscado óptimo global. Siendo nuestro algoritmo sencillo y fácil de entender.

Analizamos el problema, diseñando ciertas soluciones y descartándolas con contraejemplos hasta encontrar la que nos pareció óptima según el resultado esperado de las pruebas entregadas por la cátedra. A partir de allí, demostramos inductivamente que nuestro algoritmo es óptimo y además, desarrollamos aún más pruebas para comprobarlo. Realizamos gráficos donde verificamos la complejidad de nuestro algoritmo, la cual ha sido bastante exitosa, y la variación en cuestión de tiempos y pesos.

Para finalizar, podemos determinar que el algoritmo Greedy elegido es óptimo ya que logra llegar a la solución óptima, con el tiempo de ejecución esperado dada la complejidad temporal que posee.

6. Correcciones

6.1. ¿Por qué es un algoritmo Greedy?

Definimos como heurística el orden descendente del coeficiente generado por la división $\frac{b_i}{t_i}$ de cada batalla i.

Como lo que buscamos es minimizar la suma ponderada, debemos tener en cuenta cuándo un término puede ser "muy grande". Esto puede ser dado el peso de la batalla (propio de cada una) y también por el momento en el que termine la batalla (porque la batalla tardó mucho tiempo o la suma de todas las batallas anteriores hizo que el tiempo aumentara considerablemente). Lo que

buscamos con nuestra heurística elegida es hacer que las batallas con gran peso se realicen primero, pero sin hacer esperar mucho a las otras batallas.

El óptimo local para cada batalla será el haber hecho una batalla de gran peso, pero que no retrase mucho el resto de las batallas (por eso nuestro coeficiente elegido). De esta forma, obtendremos el óptimo global, en el que los tiempos estarán aumentando proporcionalmente a cómo decrecen los pesos, de la mejor forma posible.

6.2. ¿Todas las soluciones sin inversiones tienen el mismo valor?

Una vez que demostramos que una solución óptima no debe tener inversiones, completaremos la demostración con probar que todas las soluciones sin inversiones tienen el mismo valor. Aplicaremos la misma lógica utilizada para la primer parte de la demostración.

Partimos de un conjunto de batallas donde se encuentran 2 (B_i y B_j) con el mismo coeficiente $\frac{b}{t}$.

Si armamos 2 schedules (sin inversiones) S y K donde ordenamos dada nuestra regla Greedy y ubicamos en S a la B_i antes de la B_j y en K, B_j antes de la B_i, deberíamos observar que ese ordenamiento entre las batallas nos termina dando el mismo resultado.

Llamemos al tiempo de finalización de las batallas antes de las batallas i y j como Z. Para el schedule S, la contribución a la suma ponderada de las batallas i, j (ordenada primero B_i, luego B_j) será:

$$b_i(Z + t_i) + b_j(Z + t_i + t_j)$$

Y en el schedule K con el orden contrario en las batallas:

$$b_j(Z + t_j) + b_i(Z + t_j + t_i)$$

Calculamos la diferencia entre ambas contribuciones:

$$\begin{aligned}
&bi(Z + t_i) + bj(Z + t_i + t_j) - [bj(Z + t_j) + bi(Z + t_j + t_i)] = \\
&biZ + bi \cdot ti + bjZ + bj \cdot ti + bj \cdot tj - bjZ - bj \cdot tj - biZ - bi \cdot tj - bi \cdot ti = \\
&bj \cdot ti - bi \cdot tj
\end{aligned}$$

Sabiendo que:

$$\frac{bj}{tj} = \frac{bi}{ti}$$
$$bj.ti = bi.tj$$

La diferencia entre ambas contribuciones dará 0:

$$bj \cdot ti - bi \cdot tj = 0$$

Por lo tanto, hemos comprobado que no infiere al resultado final qué batalla viene primero (porque la suma de los términos será la misma). Como no afecta al resto de la sumatoria, dos soluciones sin inversiones, que tengan batallas con el mismo coeficiente, deben ser equivalentes.