Teoría de Algoritmos

Curso Buchwald - Genender

Trabajo Práctico 1: Algoritmos Greedy en la Nación del Fuego

23 de abril de 2024

Candela Matelica	110641
Jesabel Pugliese	110860
Yamila Shih	110521

1. Introducción

El presente informe reúne la documentación de la solución al problema propuesto para el primer trabajo práctico de la materia Teoría de Algoritmos, utilizando una metodología de resolución de tipo Greedy.

2. Presentación del problema

Es el año 10 AG, y somos asesores del Señor del Fuego (líder supremo de la Nación del Fuego). El Señor del Fuego tiene varias batallas con las cuales lidiar. 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.

3. Consigna

Se pide 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 cuál se minimiza $\sum_{i=1}^{n} b_i F_i$.

4. Resolución

4.1. Análisis del problema

En esta ocasión, nos enfocaremos en ayudar al Señor del Fuego a analizar las batallas que enfrentará, con el objetivo de minimizar $\sum_{i=1}^{n} b_i F_i$. Para comenzar, definimos las variables clave:

- $n \rightarrow$ cantidad de batallas a combatir.
- $F_i \rightarrow$ momento en el que se termina la batalla i.
- $t_i \rightarrow$ tiempo que necesita el ejército para ganar la batalla i.
- $b_i \rightarrow$ importancia de la batalla i.

La pregunta clave es: ¿Cuál es el orden óptimo para combatir las batallas? Necesitamos determinar cuál será el próximo enfrentamiento una vez que concluyamos con uno.

Planteamos diferentes criterios y a medida que descartamos soluciones y detectamos patrones, nos encaminamos hacia nuestra solución final. A continuación, procedemos a mostrar nuestro método de análisis.

Dado que la sumatoria es de la forma $\sum_{i=1}^{n} b_i F_i$, nos interesaría minimizar o bien b_i, o bien t_i a mayor i, ya que sabemos que F_i va aumentando a mayor i. Analicemos qué pasa si nos enfocamos en minimizar cada uno de estos por separado:

¿Qué pasa si ordenamos de mayor a menor importancia?

En un principio, nos inclinamos por la idea de que debíamos poner las batallas de mayor importancia b_i al principio, ya que en esta instancia el final de la batalla i F_i sería menor porque sumaría menos elementos, y a medida que avanzo en la suma ponderada b_i decrecería mientras que el F_i crecería, por lo que quisimos intentar que el menor b_i quede con la mayor F_i. Pero al llevar a cabo el análisis, nos encontramos con un contraejemplo:

Dado el vector TB que representa la duración de las batallas e importancia, respectivamente, ordenados de mayor a menor importancia:

$TB = [(20, 15), (11, 10), (9, 8)]$

Primero, calculamos todas las posibles sumas $\sum_{i=1}^{n} b_i F_i$ según todas las posibles formas de ordenar las batallas:

1. $TB = [(20, 15), (11, 10), (9, 8)]$
   $\sum_{i=1}^{n} b_i F_i = 15 \times 20 + 10 \times (20 + 11) + 8 \times (20 + 11 + 9) = 930$
2. $TB = [(20, 15), (9, 8), (11, 10)]$
   $\sum_{i=1}^{n} b_i F_i = 15 \times 20 + 8 \times (20 + 9) + 10 \times (20 + 9 + 11) = 932$
3. $TB = [(11, 10), (20, 15), (9, 8)]$
   $\sum_{i=1}^{n} b_i F_i = 10 \times 11 + 15 \times (11 + 20) + 8 \times (11 + 20 + 9) = 895$
4. $TB = [(11, 10), (9, 8), (20, 15)]$
   $\sum_{i=1}^{n} b_i F_i = 10 \times 11 + 8 \times (11 + 9) + 15 \times (11 + 9 + 20) = 870$
5. $TB = [(9, 8), (11, 10), (20, 15)]$
   $\sum_{i=1}^{n} b_i F_i = 8 \times 9 + 10 \times (9 + 11) + 15 \times (9 + 11 + 20) = 872$
6. $TB = [(9, 8), (20, 15), (11, 10)]$
   $\sum_{i=1}^{n} b_i F_i = 8 \times 9 + 15 \times (9 + 20) + 10 \times (9 + 20 + 11) = 907$

Observemos que la mínima suma se da para el valor de 870 (ítem 4), y la sumatoria para el orden propuesto nos dio 930 (ítem 1), por lo que ordenar las batallas de mayor a menor importancia no resuelve nuestro problema.

¿Qué pasa si ordenamos de mayor a menor tiempo?

Surgiendo de una idea semejante a la anterior y planteándonos la relavancia del tiempo en el cálculo de la suma ponderada, ordenamos las batallas de mayor a menor tiempo para ver cómo interfiere esto en la sumatoria respecto a nuestra anterior idea. Pero otra vez nos encontramos con un contraejemplo que muestra que este orden no minimiza $\sum_{i=1}^{n} b_i F_i$.

Dado el mismo vector TB ordenado de mayor a menor tiempo de duración de las batallas:

$TB = [(20, 15), (11, 10), (9, 8)]$

El ítem 1 de la sección anterior representaría en este caso las batallas ordenadas de mayor a menor tiempo, cuyo resultado (930) no es el de la mínima suma, calculada en el ítem 4 (870).

¿Hay otra manera?

Estos resultados nos llevaron a pensar en la importancia de la relación entre b_i y t_i para minimizar cada multiplicación $b_i F_i$ a mayor i, por lo que decidimos seguir analizando, ya que observamos que ambos afectan a la suma ponderada. Así surge nuestra idea utilizada: usar la relación entre el tiempo y la importancia correspondientes.

Si bien ambos son relevantes, decidimos que la relación tome forma $\frac{t_i}{b_i}$ y ordenar de menor a mayor. Nuestro interés es que estén las relaciones menores al principio, serían aquellas tales que t_i es comparativamente menor a b_i, por lo que el producto de estas sería menor a otros de los posibles casos. A medida que avanzamos nos encontramos con que F_i incrementa, por su forma de suma acumuluada de tiempos, por lo que tener al final los t_i y b_i que son similares haría que estos productos no sean los menores que este subíndice podría tener, pero de todas formas podríamos obtener una idea de equilibrio porque los productos seleccionados al principio tienen resultado mucho menor a lo que obtendríamos si fueran los últimos.

Retomando el ejemplo antes dado:

Ordenamos el vector de menor a mayor por su relación $\frac{t_i}{b_i}$:

$\frac{t_1}{b_1} = \frac{11}{10} = 1{,}1$

$\frac{t_2}{b_2} = \frac{9}{8} = 1{,}125$

$\frac{t_3}{b_3} = \frac{20}{15} = 1.\overline{3}$

El vector nos queda:

$TB = [(11, 10), (9, 8), (20, 15)]$

Que corresponde al orden cuya suma fue calculada en el ítem 4 y que dio como resultado 870, la mínima suma.

Aunque demostramos que para este ejemplo particular el orden de las batallas dada por la relación $\frac{t_i}{b_i}$ minimiza la sumatoria $\sum_{i=1}^{n} b_i F_i$, más adelante, en la sección de Optimalidad, explicaremos por qué esta forma de ordenarlas funciona para cualquier caso y da siempre la mínima suma.

4.2. Algoritmo

A continuación se muestra el algoritmo Greedy de solución al problema.

```
def orden_de_las_batallas(batallas):
    batallas_ord = sorted(batallas, key=lambda x: x[1] / x[2])
    F = 0
    suma_total = 0

    for i in range(len(batallas_ord)):
        F += batallas_ord[i][1]
        suma_total += F * batallas_ord[i][2]

    return batallas_ord, suma_total
```

Dado `batallas_ord` el vector de batallas ordenadas según la proporción $\frac{t_i}{b_i}$, `F` representando el momento en que se termina cada batalla recorrida y `suma_total` donde vamos guardando la suma ponderada, el código lo que hace es, de manera iterativa y recorriendo las batallas ordenadas, actualizar la sumatoria dados los F_i y b_i y finalmente devolverla junto con las batallas ordenadas.

El algoritmo propuesto es Greedy porque en cada iteración vamos seleccionando, de las batallas que nos quedan a analizar, siempre la que tiene mayor importancia por unidad de tiempo, sin considerar el resultado final de la sumatoria. Esto lleva a sucesivos óptimos locales en los que se minimiza la sumatoria dada por las batallas ya vistas, y finalmente a un óptimo global con la sumatoria total minimizada.

4.3. Complejidad

Para analizar la complejidad del algoritmo, consultamos la documentación oficial de Python sobre el método de ordenamiento `sorted`, que utiliza el algoritmo de ordenamiento TimSort. TimSort, desarrollado por Tim Peters para su inclusión en la biblioteca estándar de Python, es una variante de Merge Sort combinado con Insertion Sort. Dicha documentacion refiere a que el algoritmo TimSort tiene una complejidad, en notacion Big O, de $\mathcal{O}(n \log n)$ en el peor caso, siendo n la cantidad de elementos del arreglo.

Considerando n como la cantidad de batallas, el proceso de ordenar el arreglo de batallas tiene una complejidad de $\mathcal{O}(n \log n)$ debido al algoritmo TimSort. Luego, recorrer este arreglo tiene una complejidad de $\mathcal{O}(n)$. Las operaciones realizadas durante el recorrido son de complejidad constante, $\mathcal{O}(1)$, al igual que las operaciones de inicialización de variables y la devolución del resultado final. Por lo tanto, la complejidad del algoritmo queda expresada como:

$$\mathcal{O}(n \log n) + \mathcal{O}(1) + \mathcal{O}(n) \cdot \mathcal{O}(1) + \mathcal{O}(1) = \mathcal{O}(n \log n) + \mathcal{O}(n) = \mathcal{O}(n(\log n + 1)) = \mathcal{O}(n \log n)$$

La complejidad del algoritmo propuesto es $\mathcal{O}(n \log n)$.

La documentación utilizada para nuestro análisis se puede encontrar en el siguiente enlace: Sorted Python

4.4. Optimalidad

Dados los vectores de tiempo $[t_1, t_2, ..., t_n]$ e importancia de las batallas $[b_1, b_2, ..., b_n]$, al ordenarlos de menor a mayor según la relación $\frac{t_i}{b_i}$ con $1 \leq i \leq n$ obtenemos la solución óptima ya que de esta forma aseguramos que las batallas que proporcionan más importancia por unidad de tiempo se ubiquen al principio de la sumatoria, lo que termina por minimizar la suma ponderada final.

Ya que el criterio de ordenamiento de basa en la relación $\frac{t_i}{b_i}$, la variabilidad en los valores de t_i y b_i no afecta la optimalidad del algoritmo.

5. Mediciones

Llevamos a cabo mediciones del tiempo de ejecución del algoritmo para distintas cantidades de batallas n con un máximo de 100000, incrementando n cada 500 elementos. Para las mediciones asignamos de manera aleatoria valores para los tiempos e importancias de las batallas utilizando la función `randint` de la biblioteca `random` de Python.

Registramos los resultados en un gráfico para corroborar que se cumple la complejidad teórica del algoritmo planteado.

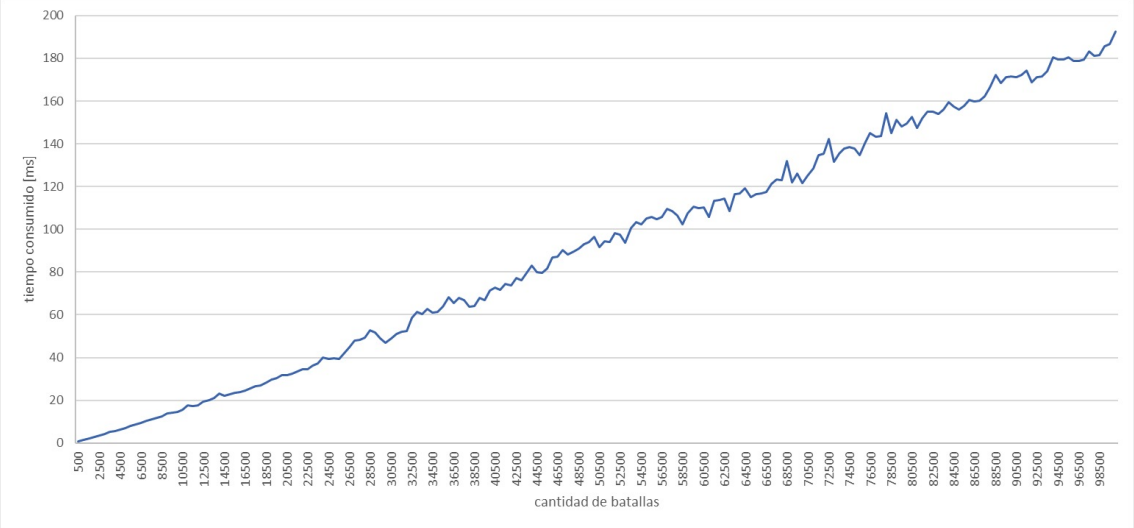

Gráfico 1: tiempo de ejecución del algoritmo en ms respecto a la cantidad de batallas. Las batallas son con t_i y b_i variables.

Habíamos dicho que la complejidad teórica de nuestro algoritmo era $\mathcal{O}(n \log n)$, así que para visualizarlo graficamos la función $n \cdot \log n$ junto con el gráfico 1.

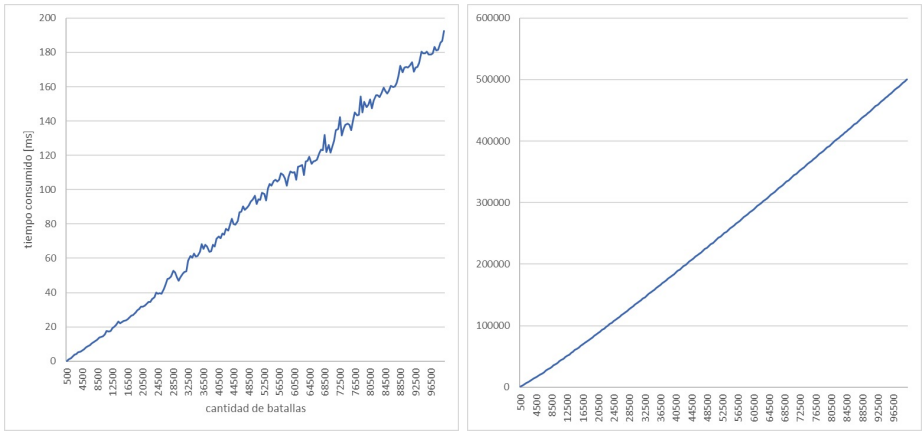

Gráfico 1 junto con el gráfico de la función $n \cdot \log n$.

Como podemos ver, ambos gráficos presentan una similitud, aunque es esperable que no sean iguales, ya que los datos de tiempos de ejecución del algoritmo son experimentales y aproximados.

Por otro lado, también realizamos mediciones para un set de datos con solo t_i variable y $b_i = 1$, y para otro con solo b_i variable y $t_i = 1$, esto para analizar cómo afectan la variabilidad de t_i y b_i a los tiempos del algoritmo planteado.

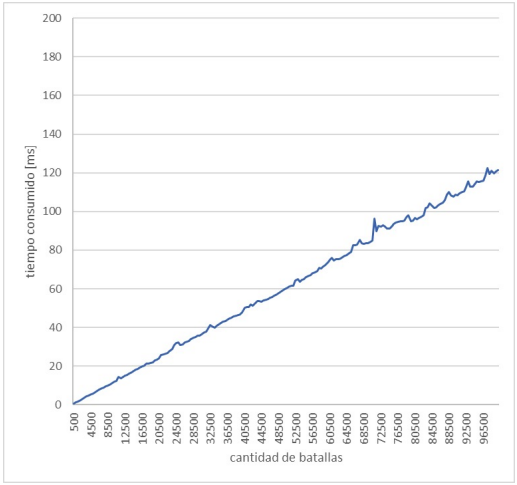

Gráfico 2: tiempo de ejecución del algoritmo en ms respecto a la cantidad de batallas. Las batallas son con t_i variable y $b_i = 1$.

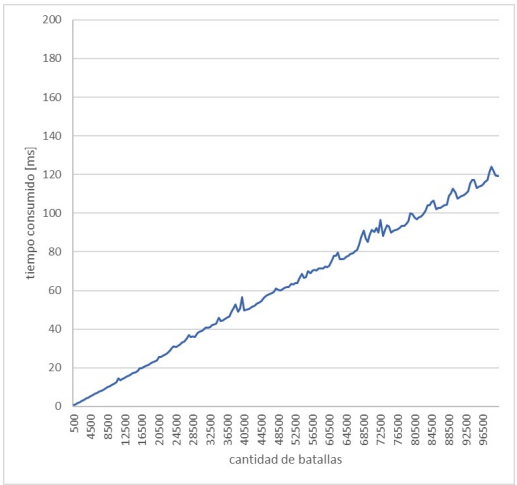

Gráfico 3: tiempo de ejecución del algoritmo en ms respecto a la cantidad de batallas. Las batallas son con $t_i = 1$ y b_i variable.

Como podemos ver, los tiempos de ejecución del algoritmo no variaron significativamente entre ambos gráficos para ninguna cantidad de batallas en específico, es decir, para toda cantidad de batallas calculada, el tiempo de ejecución del algoritmo fue parecido en ambos casos. Con esto podemos concluir que la variabilidad de los valores de t_i y b_i no afecta a los tiempos del algoritmo planteado.

6. Conclusiones

El presente trabajo nos permitió poner en práctica nuestros conocimientos sobre algoritmos Greedy e implementarlos en un algoritmo en que podamos ser capaces de visualizar su funcionamiento. Además nos permitió comparar varias opciones posibles de resolución del algoritmo y ser capaces de elegir la mejor, en base tanto a resultados experimentales como a lógica matemática.

Asimismo, realizamos análisis de optimalidad y complejidad que pudimos comprobar experimentalmente. En particular, para corroborar el cumplimiento de la complejidad teórica calculada, realizamos un gráfico que medía los tiempos de ejecución del algoritmo en base a la cantidad de batallas y lo comparamos con el gráfico de la función $n \cdot \log n$, y dado el parecido entre la tendencia de ambos gráficos, pudimos comprobar que efectivamente el algoritmo tiene una complejidad de $\mathcal{O}(n \log n)$.