.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 1
Algoritmos Greedy

17 de septiembre de 2025

Agustin Ferronato
111991

Federico Parsons
111250

Axel Alvarado
107679

Ignacio Sebastián Oviedo
109821

1. Introducción

En el presente informe se realiza el análisis de un algoritmo greedy, diseñado para encontrar la solución optima al problema que se describe a continuación.

Problema: Dado un conjunto de batallas, definidas por un peso b_i, que representa la importancia de dicha batalla, y su tiempo de duración t_i, se define a F_i como el momento de finalización de la batalla i. En caso de que la batalla i fuese la primera, entonces $F_i = t_i$. Para el resto de casos, $F_i = F_j + t_i$, si $j < i$. El objetivo es dar con un orden de batallas que logre minimizar la suma ponderada de los tiempos de finalización, dada por:

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

2. Algoritmo planteado

El algoritmo greedy planteado **encuentra siempre la solución optima**, y posee la siguiente estructura:

Algoritmo: Del listado de batallas, tomo aquella que maximiza el cociente $\frac{b_i}{t_i}$, la quito de dicho listado y la agrego al final de uno nuevo. Esto se realiza iterativamente hasta que no queden batallas en el conjunto original, quedando ordenadas de mayor a menor en la nueva lista, según dicho cociente.

El algoritmo es greedy ya que sigue una **regla básica** (tomar la batalla que maximiza el cociente, quitarla de la lista y agregarla en la nueva) que busca un **optimo local** en base a su **estado actual**. Con optimo local nos referimos a agregar aquella batalla que, dentro de las restantes (estado actual), sea la que minimiza la suma parcial obtenida. De manera intuitiva, podríamos creer conveniente el hecho de agregar la batalla que cumpla dicha regla, ya que preferimos que aquellas batallas con un peso asociado mayor (que permanece constante, independientemente del orden), sean multiplicadas por un valor menor de tiempo acumulado.

El código luce de la siguiente manera:

```python
def calculo_coeficiente(batallas: list[tuple[int, int]]) -> int:
    suma_total: int = 0
    felicidad_i: int = 0
    for batalla in batallas:
        suma_total += batalla[0] * (batalla[1] + felicidad_i)
        felicidad_i += batalla[1]
    return suma_total

# info[i] == (bi, ti)

def orden_batallas(info: list[tuple[int, int]]):
    batallas: list = sorted(info, key=lambda x: x[0] / x[1], reverse=True)
    return batallas, calculo_coeficiente(batallas)
```

Si bien no es necesario para el algoritmo en si, se calcula ademas la suma detallada en la sección anterior.

Notar que, si bien el algoritmo se describe como tomar iterativamente la batalla de mayor cociente en cada paso, lo que implicaría un costo de $O(n)$ por cada una de las n batallas y, en consecuencia, una complejidad total de $O(n^2)$, en la implementación se aprovecha el ordenamiento provisto por el lenguaje. Al ordenar la lista una sola vez según dicho cociente, con costo $O(n \log(n))$, basta luego con recorrerla secuencialmente para obtener el mismo resultado. De este modo, el ordenamiento no es lo que hace al algoritmo greedy, sino simplemente una optimización en términos de complejidad.

3. Demostración de la optimalidad del algoritmo

Sea O una solución optima del problema, con un conjunto ordenado de batallas $S_1 = \{v_1, v_2, ..., v_n\}$. Denotamos por $b(v_i)$ el peso de la batalla i-ésima y por $t(v_i)$ su tiempo de duración, para cada $i = 1, 2, \ldots, n$. Decimos que O presenta una inversión si para dos elementos contiguos v_k y v_j tales que $k < j$, se cumple que:

$$\frac{b(v_k)}{t(v_k)} < \frac{b(v_j)}{t(v_j)}. \tag{1}$$

Sean $S_2 = \{v_1, v_2, ..., v_j, v_k, ..., v_n\}$ el conjunto que no posee dicha inversión y $F(S_m)$ la función que nos devuelve el valor de la suma para S_1 y S_2.

Queremos demostrar que al intercambiar dichos elementos en nuestro conjunto S_1 la situación no puede empeorar. Es decir, que la suma ponderada luego de revertir la inversión reduce o mantiene su valor.

Desarrollamos ambas sumatorias:

$$F(S_1) = C_1 + b(v_k)\big(t(v_1) + t(v_2) + \cdots + t(v_k)\big) + b(v_j)\big(t(v_1) + t(v_2) + \cdots + t(v_k) + t(v_j)\big) + C_2$$

$$F(S_2) = C_1 + b(v_j)\big(t(v_1) + t(v_2) + \cdots + t(v_j)\big) + b(v_k)\big(t(v_1) + t(v_2) + \cdots + t(v_j) + t(v_k)\big) + C_2$$

donde C_1 y C_2 representan los términos comunes antes y después del intercambio, que permanecen constantes.

Restando y desarrollando ambas expresiones:

$$F(S_1) - F(S_2) = b(v_j)t(v_k) - b(v_k)t(v_j)$$

Ahora, a partir de la condición de inversión en (1), tenemos:

$$\frac{b(v_k)}{t(v_k)} < \frac{b(v_j)}{t(v_j)} \quad \Longrightarrow \quad b(v_k)t(v_j) \leq b(v_j)t(v_k) \quad \Longrightarrow \quad 0 < b(v_j)t(v_k) - b(v_k)t(v_j).$$

Se concluye entonces que:

$$F(S_1) - F(S_2) > 0 \quad \Longrightarrow \quad F(S_1) > F(S_2).$$

Demostramos entonces que el intercambio reduce o mantiene el valor de la suma. Aquella solución que contiene inversiones en sus elementos puede mejorarse, por lo tanto, no es optima. Entonces, al ir deshaciendo dichas inversiones eventualmente llegaríamos a la solución optima del problema, donde:

$$\frac{b(v_i)}{t(v_i)} \geq \frac{b(v_{i+1})}{t(v_{i+1})}, \quad \forall i \in \mathbb{N} \cap [1, n)$$

4. Complejidad computacional y mediciones

4.1. Justificación de la complejidad

La complejidad del algoritmo es:

$$T(n) = O(n \log(n))$$

Justificación: Sabiendo que el ordenamiento tiene un costo de $O(n \log(n))$, y que luego se recorren cada una de las batallas resultantes para el calculo del coeficiente total, costando esto ultimo $O(n)$, la complejidad resulta:

$$T(n) = O(n \log(n)) + O(n)$$

Sabiendo que:

$$\forall n \in \mathbb{N},\ n > 10 \ \Rightarrow\ n \log(n) > n$$

Y que para dos funciones f_1 y f_2 con complejidades $O(g_1)$ y $O(g_2)$ respectivamente $f1 + f2 = O(max(g1, g2))$, se concluye que:

$$T(n) = O(nlogn)$$

Como habíamos enunciado previamente.

4.2. Mediciones

Con el objetivo de verificar empíricamente la complejidad computacional, se realizaron diversas mediciones utilizando la técnica de cuadrados mínimos. Para cada ejecución, se tomaron un total de 35 muestras aleatorias, para tamaños de entrada desde $n = 1000$ hasta $n = 100000$.

Los valores de dichas muestras son tuplas de la forma (b_i, t_i), donde ambos valores son generados de forma aleatoria dentro del rango $[10, 1000]$

Ademas, con el objetivo de reducir el ruido de la mediciones, se utilizó como resultado, para cada valor de entrada de la muestra, el promedio de 10 ejecuciones.

En las figuras 1 y 2 se muestran, respectivamente, el tiempo de ejecución y el error correspondiente a una de dichas ejecuciones. El error cuadrático obtenido fue $r = 8{,}6957 \times 10^{-5}$

Si bien la complejidad se puede comprobar empíricamente utilizando este único resultado, se realizaron adicionalmente 50 ejecuciones para las 35 muestras aleatorias descritas anteriormente. De estas ejecuciones, se llego a un error cuadrático máximo de valor $r_{max} = 3{,}379 \times 10^{-4}$. Podemos concluir entonces (de manera empírica) **que la complejidad teórica coincide con la complejidad real del algoritmo**.

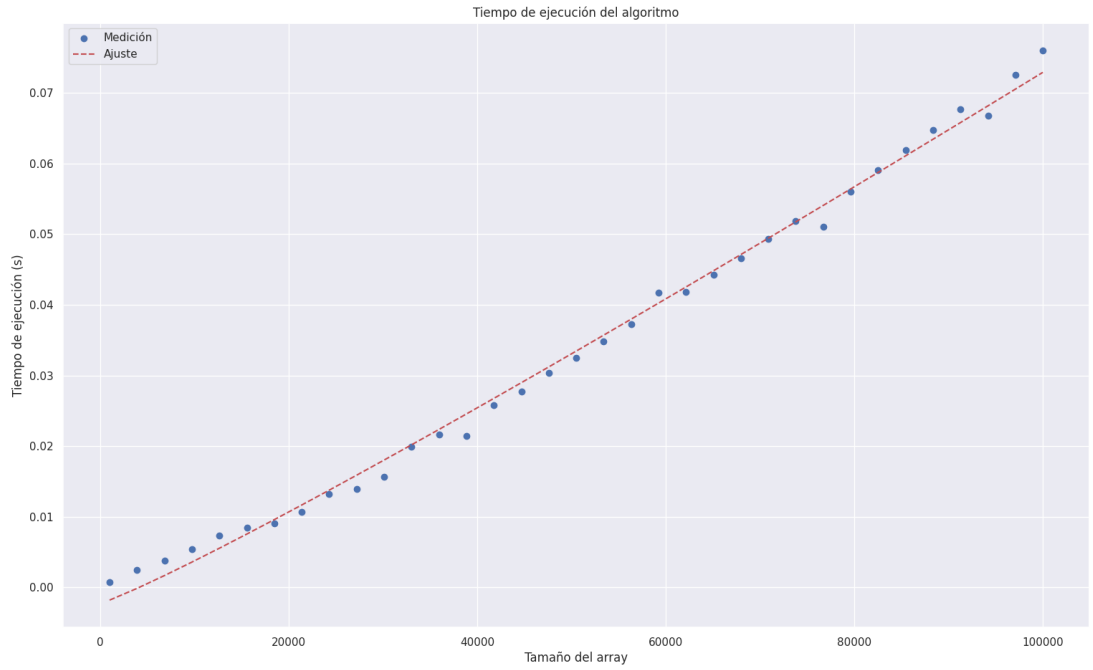

Figura 1: Tiempo de ejecución del algoritmo según el tamaño de entrada.

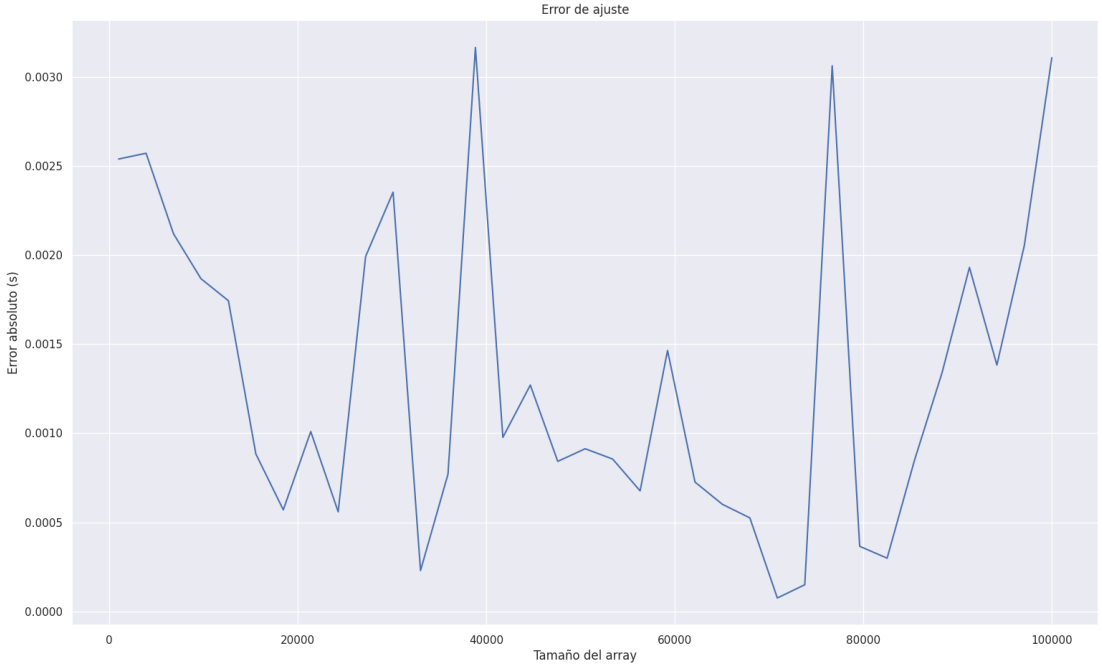

Figura 2: Error de ajuste para cada valor de entrada

A continuación, se analizó cómo varía el tiempo de ejecución en función de los valores de t_i y b_i. La metodología consistió en generar, para cada uno de los cuatro rangos considerados, un conjunto de 1000 elementos seleccionados aleatoriamente dentro del rango, calcular el tiempo de ejecución del algoritmo con ese conjunto y repetir este procedimiento 30 veces para cada punto. De este modo, para cada rango se obtuvieron 30 mediciones representadas como puntos en la Figura 3. Los resultados muestran que, dentro de los rangos analizados, las variaciones de los parámetros t_i y b_i no producen diferencias significativas en el tiempo de ejecución.

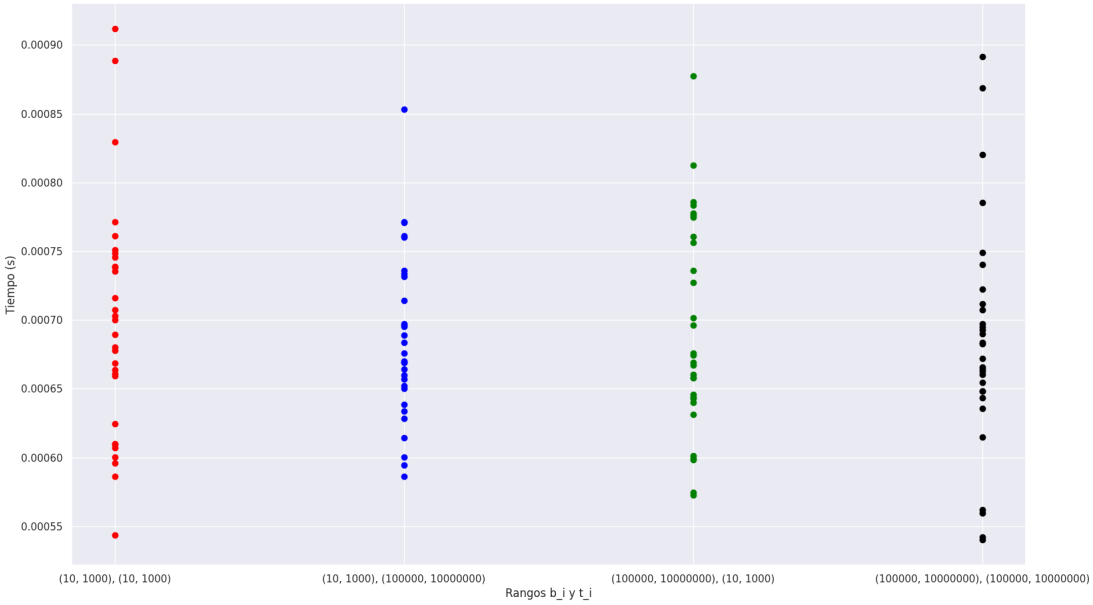

Figura 3: Tiempo de ejecución variando t_i y b_i

5. Ejecución y comparación con algoritmo de backtracking

Para comprobar que el algoritmo efectivamente devolvía la solución optima para distintas ejecuciones, se realizo la comparación de su salida contra la del algoritmo de backtracking que se encarga de resolver el mismo problema. Como este ultimo comprueba (con podas incluidas) todas las permutaciones del orden de las batallas, nos aseguramos que al final de su ejecución siempre nos devuelva la solución optima. El mismo fue codificado de la siguiente manera:

```
def coeficiente_bt (batallas, indice, solucion_actual, mejor_solucion):
    ultimo = solucion_actual.index(None) if None in solucion_actual else len(
    batallas)

    c_solucion_actual = calculo_coeficiente(solucion_actual[:ultimo])

    if indice == len(batallas):
        if c_solucion_actual < mejor_solucion[1]:
            return [solucion_actual.copy(), c_solucion_actual]
        else:
            return mejor_solucion.copy()

    if c_solucion_actual > mejor_solucion[1]: # poda
        return mejor_solucion.copy()

    batalla = batallas[indice]

    for i in range(len(batallas)):
        if solucion_actual[i] is None:
            solucion_actual[i] = batalla
            mejor_solucion = coeficiente_bt(batallas, indice + 1, solucion_actual,
    mejor_solucion)
            solucion_actual[i] = None

    return mejor_solucion
```

Como el tiempo de ejecución resulta muy alto para valores de n grandes (recordar que los algoritmos de backtracking tienen una complejidad de $O(2^n)$), nos limitamos a realizar ejecuciones de prueba de tamaño 10.

6. Conclusiones

En conclusión, se demostró que el algoritmo greedy planteado para solucionar el problema obtiene siempre la solución optima. Antes de demostrarlo formalmente, realizamos varios ejemplos de ejecución haciendo uso de un algoritmo backtracking que, por su naturaleza (verificar todas las permutaciones de los elementos de la lista de batallas), obtiene siempre la solución optima.

Ademas, realizamos varias mediciones utilizando la técnica de cuadrados mínimos para verificar empíricamente que nuestro algoritmo tiene una complejidad de $O(nlogn)$, y que la variabilidad de los valores de t_i y b_i no afecta de manera significativa al tiempo de ejecución final del algoritmo.

Asimismo, se justifico la razón por la cual nuestro algoritmo es greedy. No lo es simplemente porque ordena como se puede observar en el código, sino porque sigue una **regla sencilla** (tomar la batalla que maximiza el cociente $\frac{b_i}{t_i}$) en base a su **estado actual** (batallas restantes), en búsqueda de un **optimo local**. Con esto ultimo, nos referimos a que es conveniente quedarnos con las batallas que cuentan con un peso acumulado mayor en relación a su tiempo, ya que preferimos que ese peso (que permanece constante a lo largo del tiempo) sea multiplicado por una menor cantidad de tiempo acumulado.