Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 1
Algoritmos Greedy en la Nación del Fuego

22 de abril de 2024

Kevin Paredes
103920

Sebastian Arroyo
100728

1. Objetivo

Se necesita optimizar la cantidad de felicidad generada por el pueblo de la nación del Fuego en cada batalla en su cruzada por derrotar a todas las otras naciones, esta felicidad depende del momento en el que finalizo cada batalla, para generar la mayor felicidad posible es necesario minimizar la suma ponderada de los tiempos de finalización de las batallas, para esto se busca una solución con la técnica de diseño Greedy para obtener el orden en el que las batallas se tienen que realizar.

2. Algoritmo Greedy para optimizar la felicidad generada por la victoria de múltiples batallas

En este trabajo realizaremos el análisis teórico y empírico de un algoritmo que nos de el orden necesario de las batallas para minimizar la suma ponderada de los tiempos de finalización:

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

ti: Tiempo que necesita el ejército para ganar cada una de las batallas
Fi: El momento en el que se termina la batalla i
bi: Peso que nos define cuán importante es una batalla
Además para una batalla j que ocurre después de una batalla i tenemos que: $Fj = Fi + tj$

2.1. Algoritmo Óptimo (tiempo y peso de batalla)

Podemos decir entonces que para maximizar la felicidad la cual depende del tiempo de finalización de las batallas y el peso de importancia de cada batalla, buscamos un orden especifico en la cual minimizamos la suma ponderada (1). A continuación se muestra el código de solución del problema:

```
def merge_sort(lista):
    if len(lista) < 2:
      return lista
    else:
        middle = len(lista) // 2
        left = merge_sort(lista[:middle])
        right = merge_sort(lista[middle:])
        return merge(left, right)
def merge(lista1, lista2):
    i, j = 0, 0
    result = []
    while(i < len(lista1) and j < len(lista2)):
        if ((lista1[i][0]/lista1[i][1]) < lista2[j][0]/lista2[j][1]):
            result.append(lista1[i])
            i += 1
        else:
            result.append(lista2[j])
            j += 1
    result += lista1[i:]
    result += lista2[j:]
    return result
```

Nuestro algoritmo toma la 'n' cantidad de batallas y las ordena mediante un criterio dado por el cociente entre el tiempo que dura una batalla y el valor "B", nuestro algoritmo ordena las batallas mediante un **"merge sort"** cuya ecuación de recurrencia es:

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

Por nuestra implementación al trabajar con slicings en python tiene complejidad O(k) según la documentación siendo k el tamaño del resultado del slicing(n/2 en la primera iteración, n/4 en la segunda, etc) convergiendo esto en 2n y por lo tanto teniendo complejidad O(n)... Por teorema maestro tenemos las constantes: A = 2, B = 2 y C = 1, siendo:

1. A: Cantidad de llamados recursivos.
2. B: proporción del tamaño original con el que llamamos recursivamente.
3. O(n^c): todo lo que no son llamados recursivos.

Entonces nos queda $\log_B A = \log_2 2$ y al ser igual a C nos queda O(n^c.log n) y en particular O(n.log n). Más adelante veremos un gráfico donde podremos apreciar mejor esta tendencia.

Si bien nuestro algoritmo depende de los valores *ti* y *bi* para obtener un criterio de ordenamiento, estos **no** afectan directamente al tiempo de ejecución del algoritmo pero si la cantidad de estos, dado que al tener que hacer más comparaciones el tiempo de ejecución del algoritmo de ordenamiento será mayor. Suponemos que no puede existir un valor *bi* negativo debido a que esta variable representa el "peso"de cada batalla y no tendría sentido que este fuera negativo, la variabilidad del valor de *ti* y *bi* no afectan a la optimalidad de nuestro algoritmo dado que tomamos un cociente entre estos dos valores en los que el criterio para ordenar las batallas depende de *ti* de forma directamente proporcional y de *bi* de forma inversamente proporcional.

Este algoritmo me retorna una lista de batallas ordenado por el cociente de tiempo *ti* entre peso *bi* de cada batalla.
Para demostrar que este algoritmo es óptimo (obtiene el resultado esperado siempre) puedo utilizar una función auxiliar que calcule la suma ponderada de la lista de batallas ordenadas que calculamos con nuestro algoritmo greedy y compararlo con los resultados esperados:

```
def calculadora_coeficiente_impacto(batallas):
    batallas_ordenadas = merge_sort(batallas)
    suma = 0
    tiempo = 0
    for batalla in batallas_ordenadas:
        tiempo += batalla[0]
        suma += tiempo*batalla[1]
    return suma
```

Número de batallas	Coeficiente de impacto
10	309600
50	5218700
100	780025365
1000	74329021942
5000	1830026958236
10000	7245315862869

Cuadro 1: Coeficiente de impacto en función del número de batallas

Ejecutamos nuestro algoritmo para comparar con los resultados esperados y obtenemos:

```
>>> %Run tp.py 10
  Algoritmo para  10 batallas:
  Batallas disponibles:
  El resultado de la suma es:  309600
  El resultado es el óptimo
```

```
>>> %Run tp.py 50
  Algoritmo para  50 batallas:
  Batallas disponibles:
  El resultado de la suma es:  5218700
  El resultado es el óptimo
```

```
>>> %Run tp.py 5000
  Algoritmo para  5000 batallas:
  Batallas disponibles:
  El resultado de la suma es:  1830026958236
  El resultado es el óptimo
```

```
>>> %Run tp.py 100000
  Algoritmo para  100000 batallas:
  Batallas disponibles:
  El resultado de la suma es:  728684685661017
  El resultado es el óptimo
```

Para continuar con la demostración de que este algoritmo es óptimo podemos explorar otros algoritmos y encontrar un contraejemplo en el que no obtengamos los resultados esperados.

2.2. Algoritmo ordenando por peso de mayor a menor

```python
def merge(lista1, lista2):
    i, j = 0, 0
    result = []
    while(i < len(lista1) and j < len(lista2)):
        if (lista1[i][1] > lista2[j][1]): #Ti, Bi
            result.append(lista1[i])
            i += 1
        else:
            result.append(lista2[j])
            j += 1
    result += lista1[i:]
    result += lista2[j:]
    return result
```

Ejecutamos nuestro algoritmo para comparar con los resultados esperados y obtenemos:

```
C:\Projects\Fiuba\TDA\Tp1\TP1-TDA-Buchwald>python tp.py 10000 peso
Algoritmo para  10000 batallas:
El valor óptimo para 10000.txt es  7245315862869
El resultado del algorithmo es  9226881084991
El resultado no es óptimo
```

```
C:\Projects\Fiuba\TDA\Tp1\TP1-TDA-Buchwald>python tp.py 100 peso
Algoritmo para  100 batallas:
El valor óptimo para 100.txt es  780025365
El resultado del algorithmo es  1009743443
El resultado no es óptimo
```

```
C:\Projects\Fiuba\TDA\Tp1\TP1-TDA-Buchwald>python tp.py 10 peso
Algoritmo para  10 batallas:
El valor óptimo para 10.txt es  309600
El resultado del algorithmo es  367700
El resultado no es óptimo
```

```
C:\Projects\Fiuba\TDA\Tp1\TP1-TDA-Buchwald>python tp.py 100000 peso
Algoritmo para  100000 batallas:
El valor óptimo para 100000.txt es  728684685661017
El resultado del algorithmo es  927523607327925
El resultado no es óptimo
```

Como podemos con observar con este algoritmo no llegamos al resultado esperado y por lo tanto no es un algoritmo greedy óptimo.

2.3. Algoritmo ordenando por duración de batalla de menor a mayor

```python
def merge(lista1, lista2):
    i, j = 0, 0
    result = []
    while(i < len(lista1) and j < len(lista2)):
        if (lista1[i][0] < lista2[j][0]): #Ti, Bi
            result.append(lista1[i])
            i += 1
        else:
            result.append(lista2[j])
            j += 1
    result += lista1[i:]
    result += lista2[j:]
    return result
```

Ejecutamos nuestro algoritmo para comparar con los resultados esperados y obtenemos:

```
C:\Projects\Fiuba\TDA\Tp1\TP1-TDA-Buchwald>python tp.py 1000 tiempo
Algoritmo para  1000 batallas:
El valor óptimo para 1000.txt es  74329021942
El resultado del algorithmo es  91603127527
El resultado no es óptimo

C:\Projects\Fiuba\TDA\Tp1\TP1-TDA-Buchwald>python tp.py 5000 tiempo
Algoritmo para  5000 batallas:
El valor óptimo para 5000.txt es  1830026958236
El resultado del algorithmo es  2261739422419
El resultado no es óptimo
```

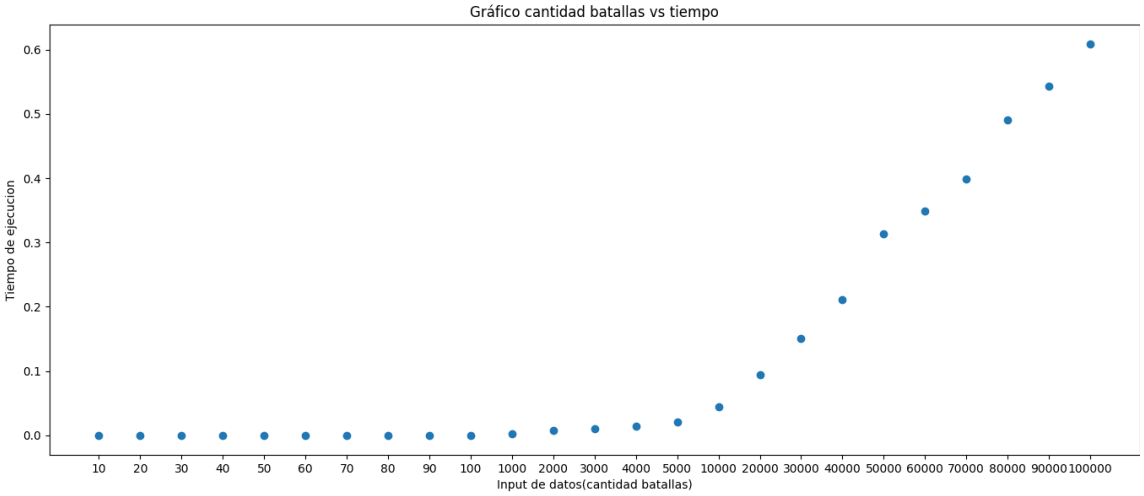
```
C:\Projects\Fiuba\TDA\Tp1\TP1-TDA-Buchwald>python tp.py 10000 tiempo
Algoritmo para  10000 batallas:
El valor óptimo para 10000.txt es  7245315862869
El resultado del algorithmo es  8926794238306
El resultado no es óptimo

C:\Projects\Fiuba\TDA\Tp1\TP1-TDA-Buchwald>python tp.py 100000 tiempo
Algoritmo para  100000 batallas:
El valor óptimo para 100000.txt es  728684685661017
El resultado del algorithmo es  896816950733118
El resultado no es óptimo
```

En este algoritmo tenemos un caso similar al algoritmo 2.2, no obtenemos los resultados esperados con este criterio para ordenar las batallas, por lo tanto tampoco es óptimo.

3. Mediciones

Se realizaron mediciones en base a crear archivos con diferente cantidad de batallas, yendo de 10 en 10 al principio hasta necesitar archivos con una cantidad de batallas cada vez mas grande para poder seguir observando la tendencia del tiempo de ejecución de este algoritmo, estos archivos se generan con números pseudoaleatorios del lenguaje(el modulo random).



Como se puede apreciar en nuestro algoritmo la tendencia del tiempo de ejecución es similar a una función $n.\log n$.

4. Conclusiones

Al tomar distintos caminos para intentar obtener la solución óptima pudimos apreciar que, por lo menos en los algoritmos greedy, es recomendable tener en cuenta en medida de lo posible la mayor cantidad de atributos en el posible algoritmo para encontrar la solución, en este caso en particular, esto fue lo que hizo que pudiéramos llegar al criterio correcto para encontrar el algoritmo deseado.

5. Corrección

Para demostrar teóricamente que este algoritmo es el óptimo podemos usar inversiones. Decimos que una batalla tiene una inversión si dentro de esa batalla tiene dos elementos bi y bj tal que $i < j$ pero $ci > cj$ donde c es el cociente entre el tiempo de duración de una batalla y su importancia, entonces: Supongamos que hay una lista de batallas en el orden óptimo que si tiene inversiones. Como tiene inversiones, debería de poder intercambiar las batallas inversibles contiguas. Al invertir estas dos, no debería de empeorar la suma ponderada total:

$$bi \in (Ti, Bi)\,, Ci = \frac{Ti}{Bi}; A = [b1, b2, ..., bi], i \in \mathbb{N}, i > 0 \tag{2}$$

$$A = [(1,1), (20,5), (10,5)] \tag{3}$$

donde A tiene una inversión porque:
$i < j : 2 < 3$ y
$C(Ai) > C(Aj) : C(A2) > C(A3) : 4 > 2$

$$A = [(1,1), (20,5), (10,5)] \tag{4}$$

ordenado según el cociente de Duración / Importancia

$$C(A) = [1, 4, 2] \tag{5}$$

A es una lista de batallas de orden óptimo con una inversión, entonces haciendo una inversión de las 2 últimas batallas la suma ponderada no debería de empeorar:

$$\sum_{i=1}^{n} b_i F_i(A) = 261 \tag{6}$$

Después de realizar la inversión de dos batallas contiguas nos queda la lista ordenada de la siguiente forma:

$$B = [(1,1), (10,5), (20,5)], C(B) = [1, 2, 4] \tag{7}$$

Y con una suma ponderada de 211 que es menor a 261, cumplimos la demostración por inversiones.
Entonces podemos decir que realizando esta inversión llegamos a la solución óptima propuesta por nosotros es decir, ordenando las batallas por el coeficiente de la duración de batalla sobre la importancia de la batalla de menor a mayor.

6. Correccion 2

Para demostrar teóricamente que este algoritmo es el óptimo podemos usar inversiones. Decimos que una batalla tiene una inversión si dentro de esa batalla tiene dos elementos bi y bj tal que $i < j$ pero $ci > cj$ donde c es el cociente entre el tiempo de duración de una batalla y su importancia, entonces: Supongamos que hay una lista de batallas en el orden óptimo que si tiene inversiones. Como tiene inversiones, debería de poder intercambiar las batallas inversibles contiguas. Al invertir estas dos, no debería de empeorar la suma ponderada total:

$$bi \in (Ti, Bi)\,, Ci = \frac{Ti}{Bi}; A = [b1, b2, ..., bi], i \in \mathbb{N}, i > 0 \tag{8}$$

$b \in (t, b)$
$A = [(t_1, b_1), (t_2, b_2), ..., (t_i, b_i)], i \in \mathbb{N}$
A' tiene inversiones entonces:

$A' = [(t_2, b_2), (t_1, b_1), ..., (t_i, b_i)], i \in \mathbb{N}$

$$\sum b_i.f_i(A) = b_1t_1 + b_2(t_1 + t_2) + b_3(t_1 + t_2 + t_3) + ... + b_i(t_1 + ... + t_i) \tag{9}$$

$$\sum b_i.f_i(A') = b_2t_2 + b_1(t_2 + t_1) + b_3(t_2 + t_1 + t_3) + ... + b_i(t_2 + ... + t_i) \tag{10}$$

para los términos posteriores a b_3 son idénticos para ambas sumatorias por lo que nos quedamos con solo los primeros 2 términos de ambas.
Comparando $\sum b_i.f_i(A)$ y $\sum b_i.f_i(A')$ para que se cumpla la condición planteada de las inversiones tenemos que $\sum b_i.f_i(A) \geqslant \sum b_i.f_i(A')$, entonces:

$$b_1t_1 + b_2(t_1 + t_2) \geqslant b_2t_2 + b_1(t_2 + t_1) \tag{11}$$

Desarrollando (11) obtenemos:
$b_1t_1 + b_2t_1 + b_2t_2 \geqslant b_2t_2 + b_1t_2 + b_1t_1$
$b_2t_1 \geqslant b_1t_2$
$\frac{t_1}{b_1} \geqslant \frac{t_2}{b_2} \Rightarrow C_1 \geqslant C_2$
$\sum b_i.f_i(A) \leqslant \sum b_i.f_i(A') \Leftrightarrow C_1 \leqslant C_2$
Entonces podemos concluir que para cualquier posible solución distinta de la propuesta por nuestro algoritmo, la suma ponderada de esta (A') va a ser mayor a la inicial (A) mientras que se cumpla que $C_1 \leqslant C_2$ que es justamente el criterio que usamos en nuestro algoritmo para obtener la solución óptima del orden de batallas.