



TEORÍA DE ALGORITMOS
(75.29) CURSO BUCHWALD - GENENDER

Trabajo Práctico 1

Algoritmos Greedy en la Nación del Fuego

7 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 finalizacion de las batallas, para esto se busca una solucion con la tecnica 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:

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

t_i : Tiempo que necesita el ejército para ganar cada una de las batallas

F_i : El momento en el que se termina la batalla i

b_i : Peso que nos define cuán importante es una batalla

Además para una batalla j que ocurre después de una batalla i tenemos que: $F_j = F_i + t_j$

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 específico en la cual minimizamos la suma ponderada (1). A continuación se muestra el código de solución del problema:

```
1 def merge_sort(lista):
2     if len(lista) < 2:
3         return lista
4     else:
5         middle = len(lista) // 2
6         left = merge_sort(lista[:middle])
7         right = merge_sort(lista[middle:])
8         return merge(left, right)
9 def merge(lista1, lista2):
10     i, j = 0, 0
11     result = []
12     while(i < len(lista1) and j < len(lista2)):
13         if ((lista1[i][0]/lista1[i][1]) < lista2[j][0]/lista2[j][1]):
14             result.append(lista1[i])
15             i += 1
16         else:
17             result.append(lista2[j])
18             j += 1
19     result += lista1[i:]
20     result += lista2[j:]
21     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 $O(n^c \cdot \log n)$ y en particular $O(n \cdot \log n)$. En el siguiente gráfico se puede apreciar un gráfico tiempo vs input de datos para poder apreciar 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 depende 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:

```
1 def calculadora_coeficiente_impacto(batallas):
2     batallas_ordenadas = merge_sort(batallas)
3     suma = 0
4     tiempo = 0
5     for batalla in batallas_ordenadas:
6         tiempo += batalla[0]
7         suma += tiempo*batalla[1]
8     return suma
```

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

Cuadro 1: Coficiente 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

```
1 def merge(lista1, lista2):
2     i, j = 0, 0
3     result = []
4     while(i < len(lista1) and j < len(lista2)):
5         if (lista1[i][1] > lista2[j][1]): #Ti, Bi
6             result.append(lista1[i])
7             i += 1
8         else:
9             result.append(lista2[j])
10            j += 1
11    result += lista1[i:]
12    result += lista2[j:]
13    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 algoritmo 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 algoritmo 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 algoritmo 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 algoritmo es 927523607327925
El resultado no es óptimo
```

Como podemos 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

```
1 def merge(lista1, lista2):
2     i, j = 0, 0
3     result = []
4     while(i < len(lista1) and j < len(lista2)):
5         if (lista1[i][0] < lista2[j][0]): #Ti, Bi
6             result.append(lista1[i])
7             i += 1
8         else:
9             result.append(lista2[j])
10            j += 1
11    result += lista1[i:]
12    result += lista2[j:]
13    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 algoritmo es 91603127527
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 algoritmo es 8926794238306
El resultado no es óptimo

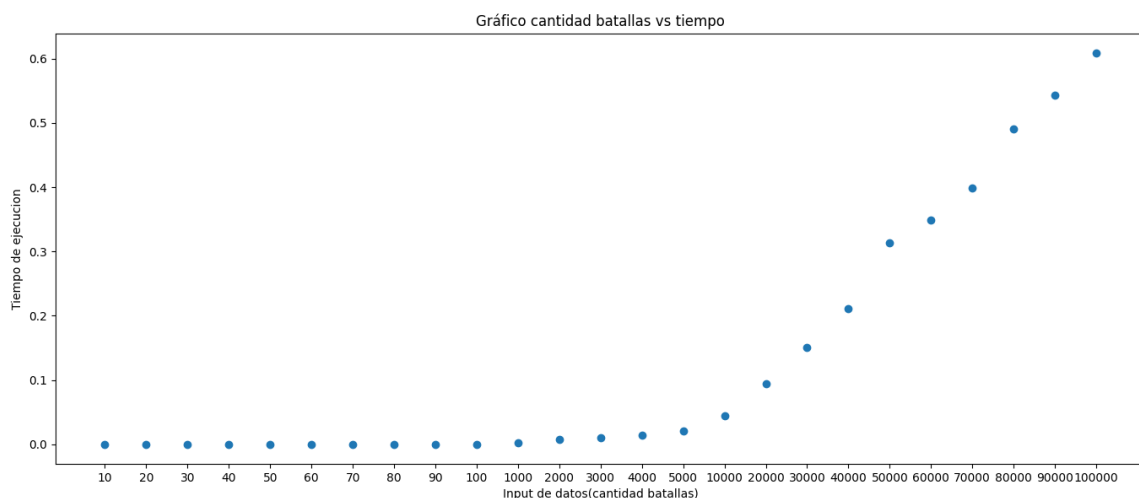
C:\Projects\Fiuba\TDA\Tp1\TP1-TDA-Buchwald>python tp.py 5000 tiempo
Algoritmo para 5000 batallas:
El valor óptimo para 5000.txt es 1830026958236
El resultado del algoritmo es 2261739422419
El resultado no es óptimo

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 algoritmo 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 numeros pseudoaleatorios del lenguaje (el modulo random) `random`).



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

4. Conclusiones

Al tomar distintos caminos para intentar obtener la solución óptima pudimos observar 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.