Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 2
Programación dinámica

2 de mayo de 2024

Leticia Figueroa
110510

Andrea Figueroa
110450

Josue Martel
110696

1. Introducción

La finalidad del presente trabajo práctico consiste en el planteamiento y análisis de un algoritmo que logre determinar cuál sería la *mejor estrategia* de ataque para la policía secreta de Ba Sing Se, los Dai Li, frente al *ataque ráfaga* que la nación del fuego efectuará.

Para el plantemiento de dicho algoritmo resulta imprescindible llevar a cabo un análisis exhaustivo del problema mismo: su forma y composición, y finalmente la ecuación de recurrencia del mismo.

Una vez consolidada nuestra propuesta de algoritmo, expondremos el análisis correspondiente al mismo, considerando factores de nuestro interés como la complejidad temporal y espacial del mismo, así como los efectos de la variabilidad de los valores dados sobre los tiempos de ejecución y sobre la optimalidad de la solución a la que llegamos.

Sobre la estrategia a obtener:

- El criterio para considerar una estrategia como *mejor* se basa en que la misma **maximice la cantidad de bajas** en las tropas enemigas.
- La estrategia resuktante debe indicar en qué minutos resulta más conveniente llevar a cabo un ataque (empleando el total de energía disponibe hasta ese momento) así como cuándo no atacar, es decir, usar ese tiempo para cargar la energía disponible para el próximo ataque.

Sobre la información disponible:

- El *ataque ráfaga* que la nación del fuego planea llevar a cabo consiste en una susesión de hordas de soldados aproximándose cada minuto, misma que durará n minutos.
- Es conocida la cantidad de soldados con los que cuenta cada horda, en cada minuto $i \in \mathbb{N} / i \leq n$. Esta cantidad la representamos como x_i.
- También conocemos qué intesidad tiene un ataque de los Dai Li, es decir, cuántas bajas podría generar en las tropas enemigas (ya que de no haber soldados, aún si se tiene una intensidad alta , no se genera ninguna baja) respecto al tiempo de carga de energía. Esta cantidad la representamos como f_i

2. Análisis del problema

En esta sección expondremos el método que seguimos para, a través de pasos lógicos, idear un álgoritmo que nos de una solución al problema propuesto.

Dado que en el problema nos informan que las batallas no se pueden realizar simultaneamente, podemos deducir que independientemente del ordenamiento escogido los tiempos de finalización siempre ascenderán; los pesos/importancias por otro lado no llevan ningún factor de acumulación repecto a la importancia de la batalla previa. Inicialmente no sabemos cómo repercuten estos datos en el cálculo de nuestra sumatoria propuesta, para ello realizamos el siguiente análisis:

NOTACIÓN: tiempo de duración (t_i), peso por batalla (b_i) y tiempo de finalización (F_i).

PROBLEMA: minimizar la suma ponderada.

$$\sum b_i F_i$$

En principio, nos interesan dos situaciones particulares que nos permitirán rescatar conclusiones importantes para resolver nuestro problema general:

2.1. Casos particulares

2.1.1. Primer caso: tiempo constante

$(t_i = cte \ \ \forall i)$

Cuando todas las batallas tienen el mismo tiempo de duración, sin importar el cronograma, la forma en la que los F_i crecen es lineal y la conocemos, veamos por ejemplo cuando $t = 1$ que los tiempos de finalización resultan:

$$F_1 = 1, \quad F_2 = 2, \quad F_3 = 3, \quad ..., \quad F_n = n$$

Es evidente que cada tiempo de finalización (F_i) es siempre superior en t respecto a la batalla anterior. Teniendo en cuenta esto, es intuitivo preocuparnos por la batalla más importante (t, b_M). Si se realiza de última estaremos empeorando el producto ponderado de dicha batalla $(F_i \times b_M)$ respecto a si esta se realizaba un orden antes en nuestro schedule exactamente en $t \times b_M$. Podemos mejorar el producto ponderado de esa única batalla mientras más antes la realicemos, si tomamos esta regla simple para el resto de las batallas restantes podemos establecer un criterio que nos lleve a una solución óptima:

Priorizar en nuestro cronograma las batallas con mayor peso, minimizando asi, el producto que tiene el mayor valor numérico b_i de entre las batallas aun no cronogramadas.

2.1.2. Segundo caso: Peso constante

$(b_i = cte \quad \forall i)$

Cuando todas las batallas tienen la misma importancia, podemos prescindir del saber el valor de este. Matemátematicamente esto se debe a que si b es constante, el término se puede extraer de la sumatoria.

$$\sum_{i=0}^{n} b_i \times F_i = b \times \sum_{i=0}^{n} F_i$$

El problema ahora se simplifica enormemente: buscamos reducir la sumatoria de tiempos de finalización ($\sum_{i=0}^{n} F_i$). Es considerablemente sencillo observar que si nos interesa minimizar el tiempo total de todas las batallas la mejor estrategia a tomar es reducir los $F_i \quad \forall i$.

Como cada F_i depende de cuándo termina la batalla previa, es esperable que (por ejemplo) para reducir el F_2 debamos buscar que tenga el menor *recargo* por la finalización de la batalla previa (F_1) que por concepto vendría siendo t_1; para las siguientes batallas buscaremos entonces retrasar lo menos posible el inicio de la batalla próxima, es decir, una batalla termina antes mientras más antes termine la batalla previa, entonces priorizaremos las batallas con menor tiempo de finalización (la mejor solución a cada paso). Formalmente, nuestro criterio resulta:

Priorizar las batallas de menor tiempo de duración, minimizando el siguiente F_i para cada batalla.

2.2. Caso general

Entendimos que hay dos lineamientos que debemos seguir:

- priorizar las batallas con menor duración.

- priorizar las batallas con mayor peso.

Es razonable buscar algún criterio que tenga en cuenta dichos lineamientos. Nuestra propuesta es imponer una relación "premio-castigo"para las variables que vimos, sean menores o mayores.

A partir de este coeficiente unificador, es que nuestro algoritmo propuesto resulta en el siguiente:

Priorizar las batallas que tengan una mejor relación entre su peso y su tiempo de duración ($k_{i(t_i,b_i)} = \frac{b_i}{t_i}$).
De forma que el ordenamiento resultante cumple: $\frac{b_i}{t_i} > \frac{b_{i+1}}{t_{i+1}} \quad \forall i / \quad 0 \leq i < n$

3. Algoritmo Propuesto

A partir del analisis llevado a cabo en la sección previa, obtuvimos la ecucación de recurrencia, que nos aproxima considerablemente a cómo sería la implementación de un algoritmo por programación dinámica que obtenga la solución óptima, ya que, la misma nos explicita la forma en la que la información relativa a las soluciones óptimas de los problemas más pequeños ($OP(k)$, $k < n$) se usan para construir la solución al problema actual $OP(n)$.

Para plasmar la lógica de la ecucación de recurrencia en código, en nuestro equipo decidimos realizar la implementación haciendo uso de un enfoque $bottom - up$, ya que este permite visualzar de forma más notoria la implementación y uso de memoization, que es el almacenamiento de los resultados de cálculos previos para evitar tener que recalcularlos. Esta técnica será de mucha utilidad para optimizar y llevar a cabo tanto la reconstrucción de las acciones que se realizan en la estrategia como para obtener el valor de la cantidad de bajas máxima.

Habiendo ya esclarecido el porqué optamos por implementar un algoritmo bottom-up, y mecionado brevemente el papel de la memorización de datos relativos a la solucion de los subproblemas, pasamos a mencionar cuál es la información de está índole que nos pareció relevante memorizar y el porqué. *En lo que sigue llamaremos "mejor baja.*[a] *el valor de la máxima cantidad posible de bajas de dicho problema, también...*

- Para hallar la mejor baja respectiva al problema de tamaño n: $OP(n)$ necesitamos saber el valor de las mejores bajas de los subproblemas previos: $OP(k)$, y como no es posible poder predecir cuales de estos valores ya no serán útiles para problemas de mayor tamaño, entonces vimos necesario el "memorizar"todas la totalidad de esos valores de los subproblemas.
  Esto se refleja en nuestro código en la creación y actualización de los valores del arreglo OP, que guarda exclusivamente el **valor de las mejores bajas obtenibles en dicho subproblema**.

- Dado que para la construcción de la solución de un problema de tamaño n cualquiera diferente de cero nos vemos en la necesidad de evaluar cúal de todas las soluciones previas (de menor tamaño) termina componiendo la mejor solución al problema actual y al final elegimos un un subproblema k que es el que logra esta maximización, vimos por conveniente recordar este dato. Pues, para la reconstrucción de la estrategia necesaria para alcanzar esta solución óptima, resulta el punto de partida el tener en cuenta que una solución óptima siempre considera que es conveniente atacar en el último minuto (respectivo al tamaño del subproblema).Entonces, resulta evidente que necesitamos saber cuál es la solución óptima respectiva a un subproblema que termina componiendo la solución a este. En este contexto resulta una optimización significativa para la reconstrucción el saber **cuál es la solución, correspondiente a un subproblema previo, que se decidió y corroboró como la mejor composición al problema que consideramos**.

  Esta observación se refleja en nuestro código en la creación y asignación de los valores del arreglo *prev_atack* con el cual resulta poco compleja el llevar a cabo la reconstrucción. De querer visualizar cómo reconstruimos la estrategia respectiva a la solución optima de un problema véase el archivo

3.1. Código

Una vez expuestas las razones de porqué decidimos almacenar ciertas informaciones relativas a las soluciones óptimas de cada subproblema, así como el porqué del enfoque mediante cual construimos nuestro algoritmo, pasamos a presentar finalmente el algoritmo propuesto.

```
1 def kills_and_strategy(n , x , f):
2     OP = [0] * (n + 1)
3     prev_atack = [0] * (n)
4     for i in range(1 , n + 1):
5         for prev in range(i):
```

```
            bajas = min(x[i - 1] , f[i - prev - 1]) + OP[prev]
            if bajas >= OP[i]:
                OP[i] = bajas
                prev_atack[i - 1] = prev
    return OP, reconstruction(prev_atack)
```

Conclusión: Probamos por absurdo que no existe un cronograma con al menos una inversión que mejore la sumatoria obtenida por nuestro algoritmo (cumple *Suposición 1*). Es decir, si no existe mejor cronograma: el propuesto es el que minimiza de mejor forma la sumatoria ponderada para cualquier conjunto de batallas con duración diferente a 0.

4. Mediciones

4.1. Primera medición

4.1.1. Procesamiento y generación de datos de entrada:

Se realizaron mediciones en base a crear arreglos para x y f de diferentes largos, yendo de 100 en 100 minutos, para los cuales, en cada caso, los elementos de x fueron generados por los valores pseudoaleatorios del lenguaje (el módulo `random`) y los elemenos de f, generados de forma creciente a pasos, igualmente, de valores pseudoaleatorios. En esta ocación no fue necesario estabilizar los resultados, el gráfico muestra de forma clara los resultados.

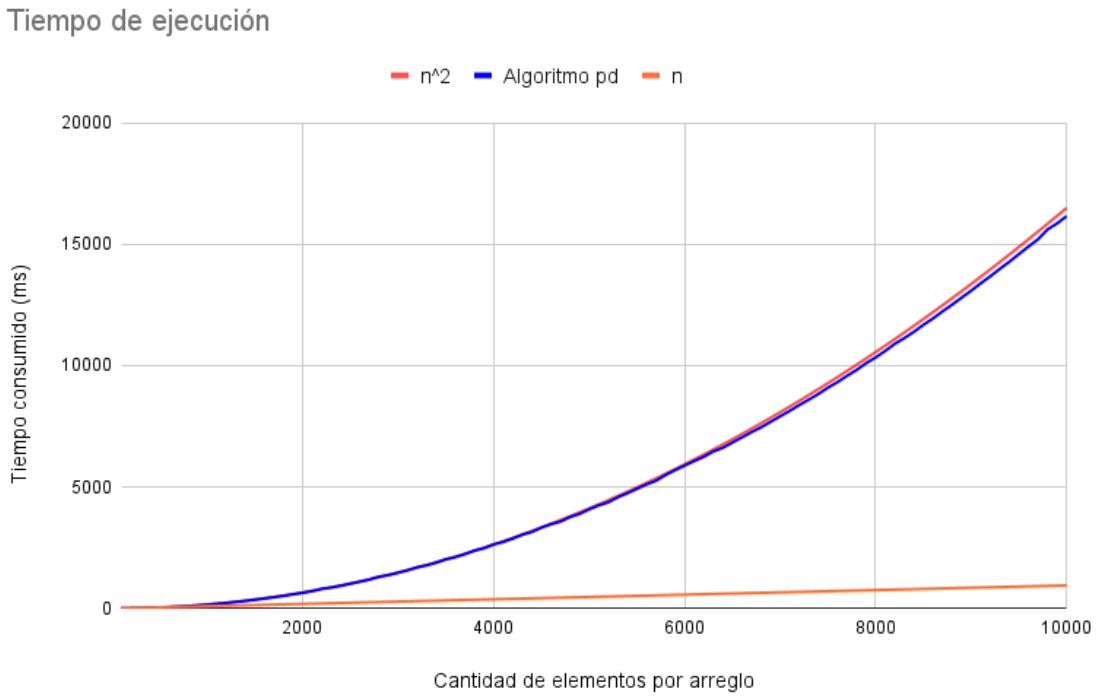

Figura 1: Medición 1

4.1.2. Análisis de los resultados:

Con estos resultados es preciso el poder afirmar que el algoritmo PD propuesto, efectivamente, tiene una tendencia **cuadrática** en función del tamaño de los datos de entrada, exactamente como lo predecimos.

4.2. Tercera medición: t_i y b_i constantes

4.2.1. Procesamiento y generación de datos de entrada:

De la misma forma se realizaron mediciones, ahora estabilizadas, en base a crear arreglos de batallas de diferentes largos, yendo de 500 en 500 batallas, para las cuales, en cada caso, el tiempo y peso por batalla fueron números constantes.

Figura 2: Medición 2

4.2.2. Análisis de los resultados:

Haciendo mediciones con $t_i = t, b_i = b \forall i$ nos encontramos con una mejora significativa en cuanto al tiempo de ejecución de nuestro algoritmo. Este es un caso en el que la variabilidad de los datos afecta los tiempos de ejecución del algoritmo. Esta diferencia se la atribuimos enteramenete al funcionamiento de *merge_sort*.

5. Conclusiones

A partir del análisis expuesto en el presente informe, podemos concluir que como asesores del Señor del Fuego, la mejor recomendación que podemos proporcionar consta de la priorización de las batallas que tengan una mejor relación entre su relevancia/importancia y su tiempo de duración. Siguiendo este criterio, obtendríamos siempre el mejor cronograma de batallas dadas las situaciones en las que estas se realizan aquellas (no se pueden realizar simultaneamente, y no hay batallas con duración 0).

Un detalle no insignificante respecto a la recomendación finalmente presentada se relaciona a los recursos que se invierten en el diseño. Vimos en la *sección 3.2* que nuestro algoritmo greedy tiene un comportamiento n-logaritmico respecto a la cantidad de datos que ha de procesar este, además de un consumo lineal de espacio (tambien respecto a la entrada). La variabilidad de los tiempos de duración e importancia de cada batalla sobre la optimalidad del algoritmo, por otro lado, termina sin ser de mayor relevancia: no observamos cambios en la eficacia al variar los datos de entrada (tiempo de duración e importancia).