Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 2
Programación Dinámica para el Reino de la Tierra

5 de mayo de 2024

Kevin Paredes
103920

Sebastian Arroyo
100728

1. Objetivo

Al ser los jefes estratégicos de los Dai Li debemos determinar en qué momentos debemos atacar para eliminar a tantos enemigos en total como sea posible. Para poder determinar esto correctamente tenemos que tener en consideración que:
Llegaran una ráfaga de soldados durante el transcurso de n minutos y en el i-ésimo minuto llegaran x_i. También sabemos que nuestros ataques tienen un alcance de soldados enemigos dada por *f(.)*, siendo esta una función monótona creciente además sabemos que si transcurrieron j minutos desde que se utilizo este ataque, entonces es capaz de eliminar hasta *f(j)*. En el caso de atacar en el *k-ésimo* minuto después de haber transcurrido j minutos se eliminara a min(x_k,f(j)) soldados

2. Algoritmo para optimizar la cantidad de soldados enemigos alcanzados por el ataque Dai Li

En este trabajo realizaremos el análisis teórico y empírico de un algoritmo que nos de la secuencia necesaria entre "Cargar Poder" y "Atacar" para poder eliminar la mayor cantidad de soldados enemigos, para ello llegamos a la ecuación de recurrencia:

$$\text{Opt}(x_n, f(n)) = \text{Opt}(x_{n-j}, f(n-j)) + min(x_n, f(j)) \tag{1}$$

n: el minuto en el que estamos
j: la cantidad de minutos desde la ultima vez que atacamos
x_n: Cantidad de soldados en el momento n
$f(n)$: Cantidad máxima de soldados a eliminar con poder n
min(x_n,f(j)): el mínimo entre x_n y $f(j)$

2.1. Análisis de la ecuación de recurrencia

Para obtener el resultado óptimo de nuestro problema vamos dividiendo en subproblemas para poder construir nuestra solución, posteriormente, del problema completo. Primero analizamos el problema como si solo tuviéramos una oleada de ataque, tenemos 2 opciones, atacar o cargar, el óptimo anterior es 0 debido a que antes a esto no había soldados y solo nos quedamos con el termino min(x_1,f(1)) entonces en este caso tenemos Opt(x_1, f(1)) = min(x_1,f(1)). Luego de obtener este óptimo analizamos con el subproblema con dos elementos, en la búsqueda de este óptimo vamos a analizar para todos los posibles valores de j usando el óptimo previo y obteniendo así la secuencia correcta para generar la mayor cantidad de soldados atacados. Luego seguimos haciendo esto de forma consecutiva hasta llegar a analizar todo nuestro problema y así poder llegar a la secuencia óptima para maximizar el valor de soldados atacados.

2.2. Algoritmo óptimo

A continuación se muestra el código de solución del problema:

```
def calcular_poder(listaSoldados,listaPoderes):
    subproblemas = [(0,"")]
    resultadoOptimo=0
    cadenaOperaciones = ""
    for i in range(len(listaSoldados)):
        for j in range(i+1):
            optimoAnterior = subproblemas[i-j][0]
            minimoActual = [listaSoldados[i], listaPoderes[j]]
            calculoMinimo = int(min(minimoActual))+optimoAnterior
            if(resultadoOptimo < calculoMinimo):
                resultadoOptimo = calculoMinimo
```

```
12                  cadenaOperaciones = str(subproblemas[i-j][1])+"C"*j+"A"
13              subproblemas.append((resultadoOptimo,cadenaOperaciones))
14      return subproblemas[-1]
```

Nuestro algoritmo recorre un arreglo de longitud n donde cada posición representa la cantidad de soldados que invaden en un minuto i y j los minutos que transcurren cada vez que atacamos a los soldados invasores, entonces $i - j$ representa el minuto del último ataque a los soldados invasores. Para nuestro análisis de la complejidad del algoritmo tenemos un bucle externo que itera un arreglo de oleadas de enemigos y otro bucle interno anidado que itera sobre el rango i + 1 el cual depende de las oleadas de soldados. Por lo tanto la complejidad del algoritmo es cuadrática O(n^2) en función de la cantidad de oleadas de enemigos.
Para analizar como afecta la optimalidad del algoritmo la variabilidad de los valores de las llegadas de enemigos y recargas podemos ver que si bien nuestro algoritmo depende de los valores xn y $f(n)$ un arreglo de oleadas de enemigos y un arreglo del poder acumulado correspondientemente, estos no afectan directamente al tiempo de ejecución pero si la cantidad de estos. También podemos ver como a cantidad igual de volumen de datos y distintos valores el tiempo de ejecución es el mismo(con el 10 y 10bis). Entonces podemos concluir que el tiempo de ejecución no varia con los valores de los soldados o de la función f(.) pero si del volumen de datos ingresados. Más adelante veremos un gráfico donde podremos apreciar mejor esta tendencia.3

Para poder demostrar que este algoritmo es óptimo usaremos la técnica de demostración por inducción. Iniciamos definiendo la estructura o forma de cada subproblema, en nuestro caso estamos tratando de maximizar la cantidad de enemigos atacados, entonces cada subproblema tiene la forma de ataques realizados en función del tiempo (minutos), sabiendo que cada ves que decidimos atacar, la fuerza con la que lo hacemos vuelve a su estado inicial y cada ves que decidimos no hacerlo ganamos mas fuerza para el siguiente encuentro. Después la forma en como los subproblemas se componen para solucionar subproblemas más grandes es:
Si decido atacar, entonces compongo la mejor solución para esa cantidad de enemigos y fuerza en el minuto que ocurrió el último ataque $i - j$ luego le sumo el mínimo entre mi fuerza actual y la cantidad de enemigos en ese encuentro. Esto se ve mejor reflejado en la ecuación de recurrencia1.
Ahora establezco la hipótesis de inducción, suponiendo que el algoritmo es óptimo para un número menor de elementos (cantidad oleadas de enemigos) demostraré que también es óptimo para un número mayor de elementos. Comienzo demostrando que el algoritmo es óptimo para el caso base $n = 1$. Para un arreglo de enemigos por minuto Xn = [271, 533, 916, etc] y un arreglo de fuerza por minuto $F(n)$ = [21, 671, 749, etc], entonces podemos extraer el subproblema base n = 1, utilizando la ecuación de recurrencia definida en 1 el seguimiento del problema es el siguiente:

$$
\begin{aligned}
\mathrm{n} &= 1 \\
\mathrm{j} &= 1 \\
\mathrm{Opt}(x_1, f(1)) &= \mathrm{Opt}(x_0, f(0)) + \mathrm{mín}(x_1, f(1)) \\
\mathrm{Opt}(x_1, f(1)) &= 0 + \mathrm{mín}(271, 21) \\
\mathrm{Opt}(x_1, f(1)) &= 21
\end{aligned}
$$

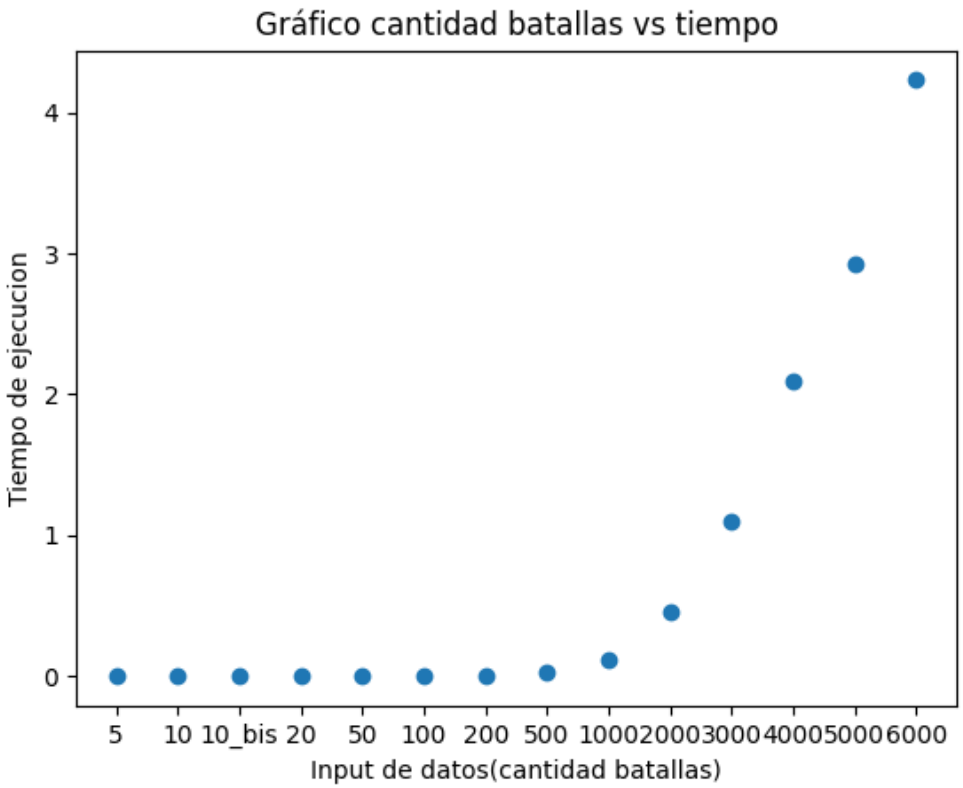
Luego suponiendo que el algoritmo es óptimo para un minuto n, puedo demostrar que también es óptimo para un minuto $n + 1$ utilizando la solución óptima para n soldados.

$$
\begin{aligned}
\text{n} &= 2 \\
\text{j} &= 1 \\
\text{Opt}(x_2, f(2)) &= \text{Opt}(x_1, f(1)) + \text{mín}(x_2, f(1)) \\
\text{Opt}(x_2, f(2)) &= 21 + \text{mín}(533, 21) \\
\text{Opt}(x_2, f(2)) &= 42 \\
\text{j} &= 2 \\
\text{Opt}(x_2, f(2)) &= \text{Opt}(x_0, f(0)) + \text{mín}(x_2, f(2)) \\
\text{Opt}(x_2, f(2)) &= 0 + \text{mín}(533, 671) \\
\text{Opt}(x_2, f(2)) &= 533
\end{aligned}
$$

Entonces para $n = 2$ y $j = 2$ el valor es el máximo y nos daría que el óptimo para $n = 2$ es 533, el cual se obtiene de cargar en el primer minuto y luego atacar. Si bien no utilizamos el valor del óptimo de $n = 1$ directamente, fue crucial para compararlo contra el caso en el que atacamos en el primer minuto y volvemos a atacar en el segundo el cual nos da el resultado 42 que corresponde a $n = 2$ y $j = 1$. Concluyendo, dado que el algoritmo es óptimo para el caso base y cada paso inductivo produce una solución óptima, el algoritmo es óptimo para todos los casos.

3. Mediciones

Se realizaron mediciones de tiempo de ejecución con distintos set de datos, usando los brindados por la cátedra y otros creados por nosotros para poder evidenciar de forma más notoria la tendencia de nuestro algoritmo. Los set de datos creados por nosotros siguieron los criterios indicados para ser considerados validos, como que los valores de f(.) tienen que ser monótonos crecientes y que la cantidad de soldados siempre es un numero positivo.



Como se puede apreciar en nuestro algoritmo la tendencia del tiempo de ejecución es similar a una función cuadrática n^2.

4. Conclusiones

Para realizar este trabajo práctico tuvimos que incorporar nuevos métodos para pensar el algoritmo con Programación Dinámica, en particular identificar cada subproblema para luego componer la solución final fue algo difícil de visualizar. Pero una vez identificado cada elemento y llegando así a la ecuación de recurrencia, pasar a código fue bastante sencillo