Teoría de Algoritmos

(75.29) Curso Buchwald - Genender

Trabajo Práctico 2

6 de mayo de 2024

Petean Marina
110564

Romano Nicolas
110830

Zielonka Axel
110310

1. Programación Dinámica para el Reino de la Tierra

1.1. Análisis del problema

El problema pide consiste en defender al Reino de la Tierra de multiples ataques rafagas de la Nacion del Fuego. El algoritmo debe encontrar el máximo de tropas enemigas que se puede derrotar, y también la secuencia de cargas y ataques correspondientes a este valor maximo. En principio se puede ver que el problema va a depender tanto de las ráfagas de enemigos, como de la función de defensa $f(.)$, ya que para un minuto i, para poder decidir en que minuto me hubiese convenido que se atacara por ultima vez se debe tener en cuenta la ráfaga de enemigos x_i de dicho minuto, así como también la función de la defensa $f(k)$, donde k son los minutos que transcurrieron desde la última defensa. Ya entonces, podríamos afirmar que es un problema de dos dimensiones y para la resolución del problema se necesitará una matriz que contenga como filas y columnas a las ráfagas de enemigos y a la función $f(.)$.

1.2. Ecuacion de recurrencia y algoritmo propuesto

1.2.1. Ecuación de Recurrencia

Definimos h como la resta entre la cantidad de rafagas (k) y el tiempo desde el ultimo ataque (j), es decir $h = k - j$. Entonces, la ecuación de recurrencia propuesta es:

$$\text{OPT}(k, j) = \max_i (\text{OPT}(h, i)) + \min(x_k, f(j)), \quad i \in [1, h]$$

Donde x_k es la cantidad de soldados provenientes de la rafaga nro k y $f(j)$ es la cantidad de soldados maximos que se puede matar con j minutos cargados

1.2.2. Primer acercamiento al algoritmo

En un primer acercamiento al problema, sospechando ya que iba a ser necesaria la utilización de una matriz para almacenar las posibilidades, ya pudimos observar que la matriz iba a ser diagonal inferior, ya que es imposible que en una ráfaga i, se use la función de defensa en un valor mayor a ese mismo i, (por ejemplo, es un absurdo plantear que en la ráfaga 4 se esté utilizando la función de defensa en 5).

1.2.3. Algoritmo para la resolución del problema

Para resolver el problema pedido, utilizamos entonces una matriz, como mencionamos previamente en **1.1**. Primero creamos una matriz cuadrada de tamaño $n + 1$ inicializando todos sus valores en 0 y un arreglo de $n + 1$ donde memorizaremos cual la carga mas conveniente para cada rafaga. Luego fuimos completando los valores siguiendo la logica de la ecuación de recurrencia. De esta forma, en cada casilla k,j nos quedaria la maxima cantidad de soldados eliminados de la ultima rafaga donde se ataco(k-j) más el mínimo entre los soldados que vienen en la ráfaga k y la cantidad que puedo eliminar con j minutos cargados. Al final del algoritmo, el maximo valor de la ultima fila sera el valor maximo de soldados que se pueden eliminar. En el arreglo $maxs$, se guarda luego de cada fila, en que columna se encuentra el máximo de dicha fila, ya que va a ser necesario para la reconstrucción de la solución.

```
1 def cargasOptimasDinamica(rafagas:list[int], n:int, f:list[int]):
2     posibilidad = []
3     for _ in range(n+1):
4         posibilidad.append([0]*(n+1))
5     maxs = [0]*(n+1)
6     for k in range(1,len(posibilidad)):
7         maximo_actual = 0
8         for j in range(1,k+1):
```

```
            posibilidad[k][j] =  posibilidad[k-j][maxs[k-j]] + min(rafagas[k] , f[j
    ])
            if posibilidad[k][j] > posibilidad[k][maximo_actual]:
                maximo_actual = j
        maxs[k] = maximo_actual
    valor_max = posibilidad[len(posibilidad)-1][maxs[len(posibilidad)-1]]
    return reconstruir_solucion(posibilidad, maxs, n), valor_max
```

Para reconstruir la solución, diseñamos un algoritmo donde empezamos por la ultima fila de la matriz y utilizando el arreglo *maxs* vamos saltando de fila donde se ataco en fila donde se ataco. Esto funciona por que la i-esima posicion *maxs* contiene cuantos minutos tendria que haberse cargado para alcanzar la maxima cantidad de soldados eliminados en la rafaga i:

```
def reconstruir_solucion(posibilidad, maxs, n):
    ataques = []
    k = len(posibilidad) - 1
    while k > 0:
        ataques.insert(0,k)
        k = k - maxs[k]

    solucion = []
    for i in range(1, j + 1):
        if i in ataques:
            solucion.append("ATACAR")
        else:
            solucion.append("CARGAR")
    return solucion
```

1.2.4. Ejecucion del algoritmo

Para ejecutar el algoritmo con un set de datos *.txt* deberas pararte a la misma altura que **algoritmo.py** y ejecutar el siguiente comando:

```
$ python algoritmo.py (PATH DEL .txt)
```

Luego te devolvera la estrategia de ataques y cargas optimas esperada junto a la cantidad de soldados eliminados correspondiente a la estrategia.

1.3. ¿Por qué es Programación Dinámica?

La solución óptima al problema global puede ser construida con soluciones óptimas a subproblemas más pequeños, ya que utilizamos los óptimos anteriores para determinar el óptimo actual. Lo podemos hacer ya que memorizamos dichas soluciónes óptimas a problemas más chicos. En nuestro caso, para una rafaga k memorizamos todos los valores posibles para j que podrian llegar, es decir, memorizamos cuantos soldados se eliminarian si en la rafaga k tenemos 1 carga o 2 cargas o ... o k cargas. Si en la rafaga k tenemos 2 cargas, significa que la ultima vez que se ataco fue en la rafaga rafaga $k-2$ y si atacamos con esas cargas, la cantidad de soldados eliminados de k con 2 minutos cargados serian la optima de la rafaga $k-2$ mas el minimo entre los que vienen x_k y los que puedo eliminar $f(2)$. Cuando hacemos referencia a *"la optima de la rafaga $k-2$"*, estamos diciendo que vino **Messi** y nos dijo con cuantas cargas tendria que haber llegado para que sea optimo, pero en realidad, como estamos iterando bottom up, la solucion de la rafaga $k-2$ ya la tenemos memorizada y solamente la usamos para crear la solucion de un problema mas grande. Por lo tanto, nuestro algoritmo cumple los requisitos para considerarse de programación dinámica.

2. Análisis de Complejidad

2.1. Complejidad temporal

La complejidad de este algoritmo se puede calcular realizando la suma de los sub-procesos que suceden dentro de él. Lo primero que se hace es crear una matriz de nXn, siendo n la cantidad de ráfagas que se van a recibir, y se la inicializa completamente en 0. La inicialización de la matriz tiene complejidad $O(n^2)$. En las filas de la matriz, van a estar representadas las ráfagas: la rágafa i está representada en la fila i (se considera que la primera ráfaga es la ráfaga numero 1, por lo que la fila 0 de la matriz está llena de ceros); y en las columnas están representadas los minutos desde que se realizó el último ataque.

Una vez hecho esto, entra en juego el algoritmo planteado, que para cada elemento de cada fila de la matriz (sin cruzar la diagonal) realiza la comparación descripta en la ecuación de recurrencia. La razón por la cual no se continúa con los elementos posteriores a la diagonal se encuentra en el punto **1.2.2**. Luego, según la columna en la que se encuentre, suma al elemento actual el máximo de la fila resultante de hacer $k - j$, siendo k la fila actual y j la columna actual. Recorrer la matriz completando cada elemento tiene complejidad $O(n^2)$, pero además se está buscando el máximo en una fila previa, y buscar un máximo en un arreglo desordenado tiene complejidad $O(n)$. Por lo que si se resolviera de esta forma, para cada elemento de la matriz estaríamos realizando una operación $O(n)$ (las comparaciones con otras celdas es acceder a un elemento de una matriz por lo que su complejidad es $O(1)$). Entonces la complejidad asciende a ser $O(n^3)$, ya que al ser operaciones que estan anidadas, sus complejidades se multiplican.

Para solucionar este problema, antes de comenzar con el proceso de búsqueda de los óptimos, creamos un arreglo llamado *maxs*, que va a encargarse de guardar la columna del máximo de valor de cada fila. Entonces, cada vez que se termina de calcular los óptimos para los elementos de cada fila, se guarda el máximo en este arreglo en la posición correspondiente a la fila que recién se terminó de recorrer. Entonces, para luego buscar el máximo en alguna fila, solo es necesario acceder a dicha posición del arreglo *maxs*, que tiene complejidad $O(1)$. Dejando entonces la complejidad total del algoritmo en $O(n^2)$

En cuanto a la reconstrucción de la solución: Es un ciclo que en el peor caso va saltando de a 1, lo que nos agrega un $O(n)$ a la complejidad del algoritmo, es decir $O(n^2) + O(n)$ que sigue siendo $O(n^2)$

2.1.1. Análisis del efecto en la complejidad de la variabilidad de los valores

Puesto que la complejidad no depende ni de f, ni de la cantidad de soldados que vienen por cada rafaga, si no de n, siendo n la cantidad de ráfágas, la variabilidad de los valores no modifica la complejidad.

2.1.2. Mediciones y gráficos

Para generar nuestros graficos utilizamos sets de datos generados de forma random, utilizando el codigo mostrado en **3.1.** Utilizamos sets de 0 a 10000 con un step de 500 rafagas y cada una con 5 pruebas para suavizar el grafico. Ademas, agregamos en cada grafico la funcion n^2 para comparar facilmente con la complejidad teorica. De igual forma, en el directorio ***graficos*** del repositorio se encuentra el codigo utilizado para arma los graficos.

Tiempos de ejecución con respecto a cantidad de batallas
Podemos observar que el gráfico sigue la forma de la función cuadrática, lo que corrobora la complejidad teórica indicada.

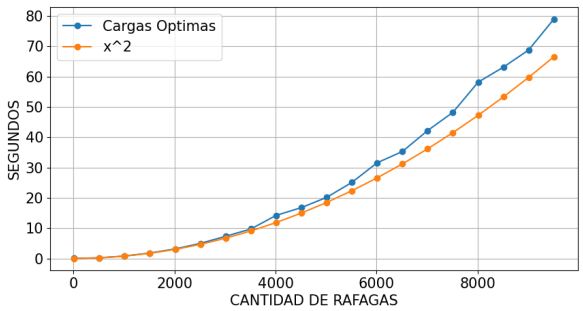

Figura 1: Gráfico de tiempos de ejecución (segundos) respecto a cantidad de rafagas (n)

Tiempos de ejecución con F constante y cantidad de soldados variables
Con este gráfico analizamos que pasa con los tiempos de ejecucion si dejamos que F(j) sea un valor constante entre 1 y n, siendo n la cantidad de rafagas. La cantidad de soldados varia entre 0 y 2 veces el valor constante de F.

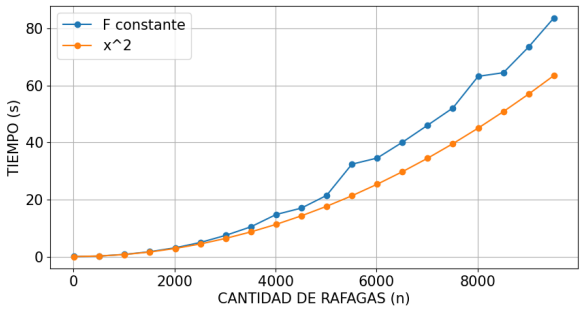

Figura 2: Gráfico de tiempos de ejecución (segundos) respecto a cantidad de rafagas (n) con F cste

Tiempos de ejecución con soldados cste
Con este gráfico analizamos que pasa con los tiempos de ejecucion si dejamos que la cantidad de soldados que vienen por rafaga sea un valor constante entre 10 y 200. La funcion F sera creciente con un maximo de 5 numeros de diferencia entre uno y otro, es decir $F(j+1) = F(j) + random(0, 5)$

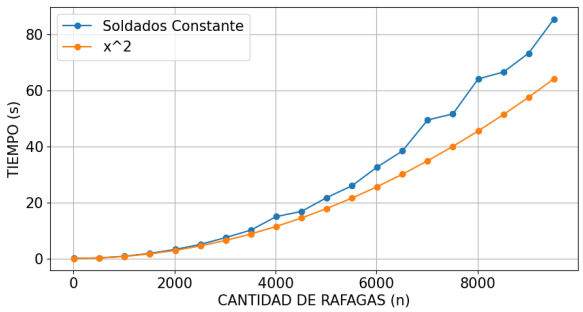

Figura 3: Gráfico de tiempos de ejecución (segundos) respecto a cantidad de rafagas (n) con soldados constante

2.2. ¿Afecta la variabilidad de los valores a los tiempos de ejecución?

No, no hemos notado cambios significativos en los tiempos de ejecución que se relacionen a la variabilidad de los valores del set de datos, como mucho, un pequeño despegue de la funcion n^2.

3. Testing

Para poder ver si nuestro algoritmo funcionaba correctamente, era necesario proveer de ejemplos de sets en donde variaban los tamaños, así como también los valores tanto del peso como del tiempo de cada batalla. En el directorio ***sets*** del repositorio de Github se pueden encontrar además de los archivos provistos por la cátedra algunos archivos con distintos sets de batallas, que varían en tamaño y en los valores de los datos.

Por otro lado, en ese mismo directorio del repositorio, está el archivo que contiene las soluciones a los ejemplos presentados por la cátedra y asi poder chequear que el resultado sea el esperado.

3.1. Generacion de set de datos y resultados esperados

Generamos nuestros set de datos a partir del siguiente algoritmo:

```
    def generar_datos_aleatorios(n, f_cste,soldados_cste):
    f = [random.randint(1, n)]
    for _ in range(n-1):
        f_anterior = f[-1]
        if f_cste:nueva_f = f_anterior
        else:nueva_f = f_anterior + random.randint(0, 5)
        f.append(nueva_f)
    if soldados_cste:
        rafagas = [random.randint(10,200)]*(n)
    else:
        rafagas = [random.randint(0,f[-1]) for _ in range(n)]
    return rafagas, n, f
```

Para nuestro propio set de datos, los resultados esperados son:
Para n = 5, el óptimo es 199
Para n = 10, el óptimo es 585
Para n = 20, el óptimo es 890
Para n = 50, el óptimo es 3245
Para n = 500, el óptimo es 52596

(el valor de n se corresponde con el nombre del archivo en cuestión y se encuentran en la carpeta ”sets”)

3.2. Ejemplo de Ejecución

Supongamos un caso de 3 ráfagas tal que: Ráfagas: 15, 30, 10 Función f: 10, 25, 30 n=3 Nuestra matriz tendra en la primera fila y columna, todos 0 representando nuestro caso base.

3.2.1. Seguimiento

El arreglo max comienza siendo [0, 0, 0]
En el arreglo max, en la posicion 0 (k-j) tendremos 0 (indice del máximo).
k=1:
Para j=1, (posibilidad[1][1])tomaremos de la matriz de posibilidad el elemento 0,0 y le sumaremos el minimo entre f para j = 1 y rafagas para k = 1 , que es 10.
En el arreglo max, en la posicion 1 tendremos entonces 1 (indice del máximo, en este caso 10). Nos

		CARGA(J)				
		0	1	2	3	
RAFAGA(K)	0	0	0	0	0	maxs
	1	0	0	0	0	[0,0,0]
	2	0	0	0	0	[0,0,0]
	3	0	0	0	0	[0,0,0]

queda entonces posibilidad[1][1]=10
maxs=[0, 1, 0, 0]

		CARGA(J)				
		0	1	2	3	
RAFAGA(K)	0	0	0	0	0	maxs
	1	0	10	0	0	[1,0,0]
	2	0	0	0	0	[0,0,0]
	3	0	0	0	0	[0,0,0]

k = 2:
Para j = 1 (posibilidad[2][1]), tomaremos de la matriz de posibilidad el elemento 1,1 y le sumaremos el minimo entre f para j = 1 y rafagas para k = 2 , que es 10.
Nos queda entonces posibilidad[2][1]=20.
Para j = 2 (posibilidad[2][2]), tomaremos de la matriz de posibilidad el elemento 0,1 y le sumaremos el minimo entre f para j = 2 y rafagas para k = 2 , que es 25.
Nos queda entonces posibilidad[2][2]=25.
maxs=[0, 1, 2, 0]

		CARGA(J)				
		0	1	2	3	
RAFAGA(K)	0	0	0	0	0	maxs
	1	0	10	0	0	[1,0,0]
	2	0	20	25	0	[1,2,0]
	3	0	0	0	0	[0,0,0]

k = 3:
Para j = 1 (posibilidad[3][1]), tomaremos de la matriz de posibilidad el elemento 2,2 y le sumaremos el minimo entre f para j = 1 y rafagas para k = 3 , que es 10.
Nos queda entonces posibilidad[3][1] = 35.
Para j = 2 (posibilidad[3][2]), tomaremos de la matriz de posibilidad el elemento 1,2 y le sumaremos el minimo entre f para j = 2 y rafagas para k = 3 , que es 10.
Nos queda entonces posibilidad[3][2] = 20.
Para j = 3 (posibilidad[3][3]), tomaremos de la matriz de posibilidad el elemento 0,2 y le sumaremos el minimo entre f para j = 3 y rafagas para k = 3 , que es 10.
Nos queda entonces posibilidad[3][3] = 10.
Finalmente tendriamos maxs=[0, 1, 2, 1]

El máximo de soldados que podemos eliminar es de 35, se obtiene de sacar el máximo de nuestra matriz en la fila k.

		CARGA(J)				
		0	1	2	3	
RAFAGA(K)	0	0	0	0	0	maxs
	1	0	10	0	0	[1,0,0]
	2	0	20	25	0	[1,2,0]
	3	0	35	20	10	[1,2,1]

4. Conclusiones

Hemos análizado nuestro algoritmo propuesto por programación dinámica para obtener el orden óptimo en el que defenderse de la nación del Fuego para maximizar la cantidad de soldados matados. Demostramos que la variabilidad de valores no afecta a los tiempos de ejecución, y no afecta a la complejidad. Por lo tanto, nuestro algoritmo de programación dinámica nos permite determinar siempre en qué momentos debemos realizar estos ataques de fisuras para eliminar a tantos enemigos en total como sea posible.