Teoría de algoritmos

(75.29) Curso Buchwald - Genender

Trabajo Práctico 2

Programación Dinámica para el Reino de la Tierra

6 de mayo de 2024

Integrantes:

- Matías Vázquez Morales (111083)
- ~~Scarlet Mendoza (108524)~~
- Nestor Fabian Palavecino Arnold (108244)

Introducción

En este informe, presentaremos un enfoque basado en programación dinámica para resolver el problema de la cantidad máxima de enemigos, y en qué tiempos corresponden dichos ataques, teniendo como datos de entrada una lista de valores que representa la cantidad de enemigos que ataca en cada minuto y una lista que representa una función de recarga de energía.

Adicionalmente, analizaremos:

- La complejidad temporal y espacial de la solución
- Variabilidad de algunos valores en el algoritmo planteado
- Tiempos de ejecución para corroborar la complejidad teórica indicada

El Algoritmo

El algoritmo planteado se encuentra en el repositorio de GitHub:

https://github.com/NestorPala/TDA-tp2/tree/main

```
def main ():
    if len(sys.argv) != 2:
        print("Ejemplo de uso: python3 tp2.py 500.txt")
        return

    path = sys.argv[1]
    filename = path.split(".")[0] + ".txt"

    enemigos_eliminados, orden_recargar_atacar = tp2_batallas_solver(path)

    escribir_resultados(filename, enemigos_eliminados, orden_recargar_atacar)

    print("\nArchivo procesado con éxito!")
    print(f"Los resultados se encuentran en el archivo solved_{filename}")
```

la función `main()`, comprueba si el número de argumentos de la línea de comandos es correcto, si no da un mensaje de cómo debe ser su uso, extrae de estos argumentos el archivo .txt que se quiere leer, hace el llamado a la función `tp2_batallas_solver` y a la función `escribir_resultados.`

La complejidad de esta función depende de estas 2 funciones, sin embargo si no las tenemos en cuenta la función `main()` tiene una complejidad constante.

```
def tp2_batallas_solver(file_path):
    with open(file_path, 'r') as file:
        lines = file.readlines()[1:]

    n = int(lines[0].strip())
    x_values = [int(x.strip()) for x in lines[1:n+1]]
    function_values = [int(x.strip()) for x in lines[n+1:]]

    return tp2(x_values, function_values)
```

abre el archivo en la ruta proporcionada en modo de lectura, lee todas las líneas del archivo omitiendo la primera línea, crea 2 listas una con los valores de X que son los enemigos y otra con el valor de las funciones, hace el llamado a la función tp2, pasando por parámetro estas 2 listas y devuelve su resultado.

En cuanto a la complejidad es O(n) donde en es el número de líneas del archivo, sin embargo, al tener otra función dentro de esta, depende de esa función.

```
def escribir_resultados(filename, enemigos_eliminados, orden_recargar_atacar):
    with open(f"solved_{filename}", 'w+') as resultados_file:
        resultados_file.write(filename)

        estrategia = ""
        for i in range(len(orden_recargar_atacar)):
            orden = orden_recargar_atacar[i]
            estrategia += orden
            if i < len(orden_recargar_atacar) - 1:
                estrategia += ", "

        resultados_file.write("\nEstrategia: " + estrategia)
        resultados_file.write("\nCantidad de tropas eliminadas: " +
str(enemigos_eliminados))
        resultados_file.write("\n")
```

Crea el archivo llamado `solved_{filename}` en modo de escritura, crea una cadena estrategia que contiene la secuencia de ordenes de carga y ataque, agregando cada cadena separada por una coma, escribe la cantidad de enemigos eliminados en el archivo.

La complejidad es O(n) donde n es el número de ordenes en `orden_recargar_atacar`

```
def tp2(x, f):
    memo = tp2_dp(x, f)
    cantidad_oleadas_enemigos = len(x)

    enemigos_eliminados, indices_solucion = reconstruir_solucion(memo)
    orden_recargar_atacar = beautify_solucion(indices_solucion,
cantidad_oleadas_enemigos)

    return enemigos_eliminados, orden_recargar_atacar
```

Hace el llamado a la función `tp2_dp` con las listas x y f como argumentos, calcula la cantidad de oleadas de enemigos, que es la longitud de la lista x, llama a las funciones `reconstruir_solucion`, `beautify_solucion` y devuelve los enemigos eliminados y el orden de recargar y atacar.

La complejidad de la función sin tener en cuenta a las otras 2 funciones llamadas es constante, ya que no tiene bucles ni estructuras de datos que crezcan con el tamaño de entrada, pero como hace el llamado a otras 2 funciones, la complejidad depende de estas.

```
def tp2_dp(lista_xi, lista_fj):
    n = len(lista_xi)
    OPT = inicializar_memo(n)
    maximos_batallas_anteriores = []

    for minuto_actual in range(1, n+1):
        maximo_batalla_actual = -math.inf

        for minuto_origen in range(n):
            if minuto_actual <= minuto_origen:
                continue

            if minuto_origen == 0:
                maximo_batallas_anteriores = 0
            else:
                maximo_batallas_anteriores =
maximos_batallas_anteriores[minuto_origen - 1]

            minuto_actual_ = minuto_actual - 1

            j = minuto_actual_ - minuto_origen
            ataque_actual = min(lista_xi[minuto_actual_], lista_fj[j])
```

```
            # ecuacion de recurrencia: OPT[i][j] = max(OPT[i-1][k] ∀ k ∈
{0,...,j-1}) + min(X[i], f(j))
            # X = lista de cantidad enemigos en el minuto i ("lista_xi")
            # f = función de recarga ("lista_fj")
            abatidos_batalla_actual = maximo_batallas_anteriores +
ataque_actual
            OPT[minuto_actual_][minuto_origen] = abatidos_batalla_actual

            # optimización O(n^3) -> O(n^2):
            # guardo el valor de la rama de valor máximo para cada minuto
            # al mismo tiempo que proceso los minutos en la matriz
            if abatidos_batalla_actual > maximo_batalla_actual:
                maximo_batalla_actual = abatidos_batalla_actual

        maximos_batallas_anteriores.append(maximo_batalla_actual)

    return OPT
```

inicializa una matriz OPT y una lista `maximos_batallas_anteriores`, itera sobre cada minuto desde 1 hasta n que es la longitud de las listas xi y fj, para cada minuto itera sobre los minutos anteriores, calcula el ataque actual como el mínimo entre el valor actual en `lista_xi` y el correspondiente en `lista_fj`, actualiza la matriz OPT con el máximo de las batallas anteriores más el ataque actual y devuelve esta matriz OPT.

La complejidad de esta función en O(n^2) donde n es la longitud de `lista_xi` y `lista_fj`, esto porque son 2 bucles anidados que recorren hasta n elementos

```
def reconstruir_solucion(memo):
    enemigos_eliminados = max(memo[len(memo) - 1])
    solucion = [ len(memo), ]

    indice = len(memo)
    while indice > 0:
        indice = indice_elemento_maximo(memo[indice - 1])
        if indice > 0:
            solucion.append(indice)

    return enemigos_eliminados, list(reversed(solucion))
```

encuentra el numero máximo de enemigos eliminados, que es el máximo valor en la última fila de la matriz memo, inicializa una lista solución con el tamaño de la matriz memo, comienza un bucle desde el final de la matriz hasta el principio, en cada iteración, encuentra el índice del elemento máximo en la fila actual de la matriz usando la función `indice_elemento_maximo()`, si el índice es mayor que 0, lo agrega a la lista solución, y devuelve el número máximo de enemigos eliminados y la lista solución en orden inverso.

La complejidad de esta función es O(n), donde n es la longitud de memo, esto es porque la recorre desde el final hasta el principio, pero al hacer el llamado a la función `indice_elemento_maximo()`, la complejidad va a depender de esa función.

```python
def beautify_solucion(indices_solucion, cantidad_oleadas_enemigos):
    beautified = []
    for i in range(1, cantidad_oleadas_enemigos + 1):
        if i in indices_solucion:
            beautified.append("Atacar")
        else:
            beautified.append("Cargar")
    return beautified
```

inicializa una lista vacía `beautified`, itera sobre cada número desde 1 hasta `cantidad_oleadas_enemigos`, en cada iteración, comprueba si el número actual está en la lista `indices_solucion`, si el numero esta, agrega a la cadena "Atacar" a la lista `beautified`, si no está agrega a la cadena "Cargar", devuelve la lista `beautified`.

La complejidad de esta función es O(n^2), donde n es la cantidad de oleadas de enemigos, esto es porque el bucle recorre desde 1 hasta `cantidad_oleadas_enemigos` y la operación in que se realiza en cada iteración tiene una complejidad O(n) en el peor de los casos.

```python
def indice_elemento_maximo(lista):
    maximo = lista[0]
    indice_maximo = 0

    for i in range(len(lista)):
        if lista[i] > maximo:
            maximo = lista[i]
            indice_maximo = i

    return indice_maximo
```

inicializa máximo con el primer elemento de la lista y `indice_maximo` con 0, itera sobre cada elemento en la lista, cada iteración, si el elemento actual es mayor que máximo, actualiza máximo con el elemento actual y `indice_maximo` con el índice actual, devuelve `indice_maximo`.

La complejidad de esta función es O(n), donde n es la longitud de la lista.

```
def inicializar_memo(n):
    memo = []
    z = -1
    for i in range(n):
        memo.append([])
        z += 1
        for j in range(n):
            memo[z].append(0)
    return memo
```

inicializa una lista vacía memo y una variable z con -1, itera sobre cada número desde 0 hasta n-1, en cada iteración, agrega una nueva lista vacía a memo e incrementa z en 1, después en un bucle anidado, agrega 0 a la lista en la posición z de memo n veces, y devuelve memo.

La complejidad de esta función es $O(n^2)$ donde n es el argumento de entrada de la función, esto es por los 2 bucles anidados que recorren hasta n elementos.

En conclusión, la complejidad total del algoritmo es $O(n^2)$.

La variabilidad de los valores de las llegadas de enemigos y recargas no afecta directamente a la complejidad del algoritmo, ya que esta se determina por el número de operaciones que tiene que realizar y no por los valores específicos de estos.

La complejidad del algoritmo es $O(n^2)$ donde n es la cantidad de oleadas de enemigos, esto se mantiene independientemente de los valores de las llegadas de enemigos y recargas. Sin embargo, si los valores son muy grandes, las operaciones aritméticas pueden llevar más tiempo, además si son valores muy variados, el algoritmo también aumenta el tiempo de ejecución.

Optimalidad del algoritmo y variabilidad de los valores de entrada

La ecuación de recurrencia de nuestro algoritmo es la siguiente:

$$OPT[i,j] = \max(OPT[i-1,k]) + \min(X[i], f(j))$$
$$k \in \{1, \ldots, j-1\}$$

Como se puede observar, no hay restricción en principio sobre que valores puede o no tomar cada casillero de la matriz de memorización. Si embargo, empíricamente podemos demostrar que, si existen valores negativos en la lista de oleadas de enemigos, la solución obtenida no es la óptima. A continuación, mostramos un ejemplo:

```
    x = [-271,-533,-916, 656, -664]
    f = [ 21,671,749,833,1543]
```

La estructura de memoización contiene los siguientes valores al momento de finalización de nuestro algoritmo:

`[[-271, 0, 0, 0, 0], [-533, -804, 0, 0, 0], [-916, -1187, -1449, 0, 0], [656, 385, 123, -895, 0], [-664, -935, -1197, -1580, -8]]`

Además, se muestra por pantalla:

`Enemigos eliminados:  -8`

`Orden de recarga/ataque:  ['Cargar', 'Cargar', 'Cargar', 'Atacar', 'Atacar']`

La cantidad de enemigos figura como `-8` porque, al momento de la reconstrucción de la solución se está tomando el elemento máximo de la última fila y a partir de ese punto aplicamos iterativamente el proceso de descubrimiento de la rama de la solución.

Esto con elementos únicamente positivos siempre funciona, pero en el caso de que hubiese valores negativos posibles para indicar la cantidad de enemigos (cosa que no tiene sentido para el dominio del problema planteado), este programa no da la solución óptima, ya que para valores de enemigos negativos, el crecimiento de la cantidad acumulada de los enemigos abatidos no es monótono, por lo cual podría suceder lo del ejemplo, que se tome como minuto de partida el máximo de la última fila (-8), en vez de tomar el valor máximo de la anteúltima fila (656).

Luego, para cantidades de enemigos 0 o positivas, el algoritmo siempre ofrece la solución óptima, ya que la función de la cantidad de enemigos abatidos es monótona creciente, con lo cual siempre podemos arrancar desde el máximo de la última fila de la matriz de memoización. En otras palabras, siempre vamos a tener que "atacar" en la última batalla.

Ejemplo de ejecución

Los ejemplos dados por la catedra, se pueden obtener los resultados pasándolos por el algoritmo, todos estos, nos dieron los resultados esperados.

Tomando como ejemplo los siguientes valores

N	X	F( )
5	125	316
	378	429
	492	563
	689	907
	274	831

Vamos a calcular cual es el óptimo, para eliminar a la mayor cantidad de enemigos posibles.

Para esto vamos a calcular todos los resultados óptimos, partiendo como si tuviéramos 1 batalla hasta N = 5.

Para 1 batalla:

Calculamos el mínimo entre X1 y F(1), en este caso 125.

125

Para 2 batallas:

125	441

Para 3 batallas:

125	441	757

Para 4 batallas:

125	441	757	1073

Para 5 batallas:

125	441	757	1073	1347

Para las 5 oleadas de enemigos que vienen, como se ve en el cuadro nos conviene atacar en todos los minutos, dando como resultado el máximo que es 1347 enemigos que se eliminaron

```
5.txt
Estrategia: Atacar, Atacar, Atacar, Atacar, Atacar
Cantidad de tropas eliminadas: 1347
```

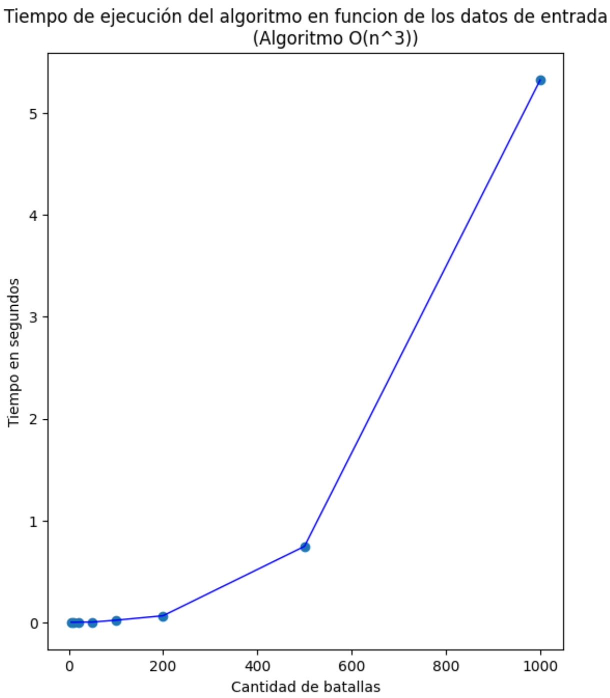
Dando como resultado el mismo que da el algoritmo planteado.

Mediciones

Complejidad del algoritmo sin la optimización de máximo valor por minuto

Orden temporal: $O(n^3)$

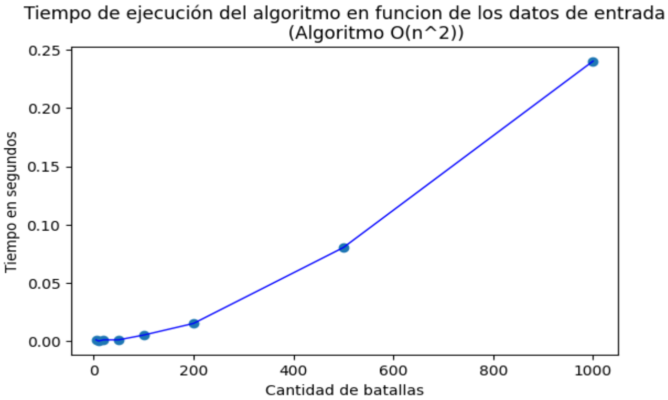
Cantidad de batallas	Tiempo en segundos
5	0.00099945068359375000
10	0.00000000000000000000
20	0.00150609016418457031
50	0.00203466415405273438
100	0.02012777328491210938
200	0.06371688842773437500
500	0.74523496627807617188
1000	5.32637214660644531250
5000	621.24419236183166503906



Complejidad del algoritmo con la optimización de máximo valor por minuto

Orden temporal: $O(n^2)$

Cantidad de batallas	Tiempo en segundos
5	0.00099754333496093750
10	0.00000000000000000000
20	0.00099873542785644531
50	0.00100183486938476562
100	0.00507354736328125000
200	0.01508188247680664062
500	0.08023452758789062500
1000	0.24034714698791503906
5000	6.27057290077209472656



Las 2 primeras corresponden a la ejecución del algoritmo sin la optimización del valor máximo por minuto. En este caso la complejidad teórica esperada es de $O(n^3)$. Donde en es la cantidad de oleadas de enemigos. Estas mediciones dan una visión de cómo se comporta el algoritmo sin ninguna optimización.

En las otras 2, son de la ejecución del algoritmo con la optimización del valor máximo por minuto implementada. Con esta la complejidad esperada se reduce a $O(n^2)$.

Para cada conjunto de mediciones, se generaron diversos sets de datos para probar el algoritmo bajo diferentes condiciones.

Estas mediciones son solo una aproximación y pueden variar dependiendo de varios factores, como la capacidad de la maquina en la que se ejecutan y la carga del sistema en el momento de la prueba.

Estos datos proporcionan una buena indicación de la eficiencia del algoritmo y como escala dependiendo del tamaño de la entrada.

Los gráficos se realizaron utilizando `matplotlib`, disponible en Python. En éstos se puede observar la complejidad temporal de nuestro algoritmo por PD.

Los datasets fueron generados utilizando el código del archivo **`generator.py`** del proyecto

Conclusiones

En conclusión, este informe ha presentado un enfoque basado en programación dinámica para resolver el problema de maximización de ataque para sucesiones de batallas.

Vimos que el algoritmo que resuelve el problema tiene originalmente una complejidad temporal cúbica, pero se puede optimizar para que sea cuadrática obteniendo el máximo valor de una fila durante el proceso de cálculo. También vimos que este pequeño cambio definitivamente afecta positivamente a los tiempos de ejecución de nuestro programa. A su vez la complejidad espacial es cuadrática y la complejidad temporal de la reconstrucción de nuestra solución también es cuadrática. Todo esto lo mostramos utilizando nuestros sets de prueba.

Comentario final: la ecuación de recurrencia sugiere que se podría (de alguna manera) optimizar el algoritmo para que la resolución del problema se haga en tiempo lineal, ya que de cada fila de la memo-matriz solo tomamos el máximo, sin embargo, esto no se puede hacer ya que para obtener el máximo sí o sí debemos calcular los demás valores, con lo cual el problema no puede ser menor a cuadrático.