Teoría de Algoritmos
(TB024) Curso Buchwald - Genender

Trabajo Práctico 2
Programación Dinámica para el Reino de la Tierra

6 de mayo de 2024

Alan Ramiro Cueto Quinto
104319

Carlos Matias Sagastume
110530

Thiago Fernando Baez
110703

1. Introducción

En este trabajo práctico tenemos el objetivo de programar un algoritmo haciendo uso de la técnica de diseño "Programación dinámica". Ba Sing Se, la capital del Reino de la Tierra, se enfrenta a una inminente amenaza: un ataque masivo de la Nación del Fuego. Los Dai Li, la fuerza de seguridad especializada de la ciudad, están a cargo de defenderla utilizando una combinación de habilidades de artes marciales. Utilizando técnicas de Tierra-control, los Dai Li han logrado detectar los planes de la Nación del Fuego para llevar a cabo un asalto en ráfagas. Durante un período de n minutos, se espera que lleguen oleadas de soldados enemigos a la ciudad, con x_i representando el número de soldados que llegarán en el i-ésimo minuto. Sin embargo, los Dai Li tienen una ventaja: la capacidad de utilizar ataques de fisuras, poderosos golpes que pueden destruir a los soldados enemigos acumulando energía durante cierto tiempo. La efectividad de estos ataques está determinada por una función $f(j)$, que indica cuántos soldados pueden ser eliminados después de acumular energía durante (j) minutos. Cuando se decide utilizar un ataque de fisuras en un momento específico, k-ésimo minuto, y han transcurrido (j) minutos desde el último uso, solo se eliminará un número de soldados igual al mínimo entre la cantidad de soldados que llegan en ese minuto x_k y la capacidad de la fisura $f(j)$. Después de usar un ataque de fisuras, se agota toda la energía acumulada. El problema se reduce a determinar cuándo usar los ataques de fisuras para maximizar el número total de enemigos eliminados. Utilizando esta información, podemos establecer una estrategia óptima para decidir cuándo realizar los ataques de fisura a lo largo de los n minutos del ataque enemigo.

2. Algoritmo para encontrar el máximo

```python
SALTO_DE_PAGINA = "\n"
POS_MINUTO_0 = 0
MINUTO_INICIAL = 1
MAXIMO_INICIAL = 0


def pd(archivo):
    minutos, oleadas, valoresFuncion = leer_archivo(archivo)
    valoresFuncion = optimizar_funcion(valoresFuncion, oleadas)
    optimo_minuto = [POS_MINUTO_0]
    padres = {}
    for i in range(MINUTO_INICIAL, minutos + 1):
        maximo = MAXIMO_INICIAL
        for j in range(i):
            valorF = valoresFuncion[j]
            oleada = oleadas[i-1]
            eliminados = min(valorF, oleada) + optimo_minuto[i - j - 1]
            if eliminados >= maximo:
                padres[i] = i - j - 1
                maximo = eliminados
            if valorF >= oleada:
                break
        optimo_minuto.append(maximo)
    return optimo_minuto[minutos], reconstruir_estrategia(padres, minutos)

def optimizar_funcion(valores_funcion, oleadas):
    maximo_enemigos = max(oleadas)
    funcion_optimizada = []
    for valor in valores_funcion:
        funcion_optimizada.append(valor)
        if valor >= maximo_enemigos:
            break
    return funcion_optimizada

def reconstruir_estrategia(padres, minuto):
    res = []
    minutos_en_que_atacan = set()
    minutos_en_que_atacan.add(minuto)
    actual = padres[minuto]
    while actual > 0:
        minutos_en_que_atacan.add(actual)
        actual = padres[actual]
    for i in range(1, minuto + 1):
        if i in minutos_en_que_atacan:
            res.append("Atacar")
            continue
        res.append("Cargar")
    return ", ".join(res)


def leer_archivo(archivo):
    oleadas = []
    funcion = []
    with open(archivo) as arch:
        arch.readline()
        n = int(arch.readline().rstrip(SALTO_DE_PAGINA))
        for i in range(n):
            oleadas.append(int(arch.readline().rstrip(SALTO_DE_PAGINA)))
        for i in range(n):
            funcion.append(int(arch.readline().rstrip(SALTO_DE_PAGINA)))
    return n, oleadas, funcion
```

En este algoritmo lo que hacemos es asumir que tenemos la estrategia optima para los n-1 minutos anteriores, arreglo al cual llamaremos Opt (cuyo caso base es Opt[0] $= 0$ ya que en el minuto 0 todavia no se realizo ningun ataque por lo tanto no hubieron bajas enemigas) y sea x la cantidad de enemigos en el minuto n, entonces para el minuto n tenemos que ver que evaluar entre las posibles cargas que puede tener el ataque en n y evaluar cual es el mejor, siendo para 0 carga la formula $c_i = min(f(i), x) + Opt[n - i]$. Opt[n] siendo n el ultimo minuto, nos dara el optimo que buscamos, ya que es aquel optimo que maximiza las bajas en toda la duracion de la batalla. y lo que nosotros queremos encontrar es el mayor de los c_i por lo tanto la ecuacion de reccurencia para obtener el maximo c_i para cada n es: $Opt[i] = max(c_i), 1 \leq i \leq n$. Este algoritmo es de programacion dinamica ya que de manera inductiva se busca resolver el problema, en este caso de forma bottom-up.

2.1. Complejidad

La complejidad del algoritmo propuesto para maximizar el número total de enemigos eliminados es $\mathcal{O}(n^2)$. Sin incluir la lectura de datos desde el archivo, hay dos bucles for: el primero, que itera según la cantidad de minutos n en los que se aparecen las oleadas de enemigos, y el otro, se ejecuta en promedio $(\frac{n}{2})$ veces. Dentro del segundo for, solo hay operaciones constante $\mathcal{O}(1)$, dando como resultado, una complejidad de $\mathcal{O}(n^2)$, siendo n los minutos. Esto se debe a que siendo T(n) el tiempo que toma el algoritmo: T(n) = T(n - 1) + $\mathcal{O}(n)$ Y como el resultado de esta ecuacion de recurrencia es: $\sum_{i=1}^{i} i = n(n+1)/2 = n^2/2 + n/2$ y esto acotado nos da que el algoritmo es de complejidad $\mathcal{O}(n^2)$. Además las funciones auxiliares para reconstruir la estrategia, leer el archivo y optimizar los valores de la función dada son todos lineales por lo que no afectan a la complejidad final.

```
def pd(archivo):
    minutos, oleadas, valoresFuncion = leer_archivo(archivo)
    valoresFuncion = optimizar_funcion(valoresFuncion, oleadas)
    optimo_minuto = [POS_MINUTO_0]
    padres = {}
    for i in range(MINUTO_INICIAL, minutos + 1):
        maximo = MAXIMO_INICIAL
        for j in range(i):
            valorF = valoresFuncion[j]
            oleada = oleadas[i-1]
            eliminados = min(valorF, oleada) + optimo_minuto[i - j - 1]
            if eliminados >= maximo:
                padres[i] = i - j - 1
                maximo = eliminados
            if valorF >= oleada:
                break
        optimo_minuto.append(maximo)
    return optimo_minuto[minutos], reconstruir_estrategia(padres, minutos)
```

3. Variabilidad de valores

3.1. Tiempo del algoritmo

La variabilidad en los valores de las llegadas de enemigos y las recargas si puede afectar el rendimiento del algoritmo:

Valores de las llegadas de enemigos (oleadas): Si las oleadas de enemigos son valores chicos, es decir, si los valores x_i son pequeños en general, es probable que el algoritmo encuentre rápidamente una solución óptima, ya que la funcion rapidamente se volvería mayor a la cantidad de enemigos por oleada y los ciclos que encuentran el máximo para cada minuto serían más cortos.Si la cantidad de los minutos n son muchos, como en el caso de prueba de los 5000 minutos de la cátedra, se puede apreciar como el algoritmo tarda más que con pocos casos, esto último ocurre porque la complejidad temporal del algoritmo es cuadrática respecto al tamaño de la entrada. Pero si hay oleadas inesperadamente grandes en momentos específicos, podría ser difícil para el algoritmo encontrar la estrategia óptima para maximizar la eliminación de enemigos ya que deberia iterar a traves de mas valores de la función.

Valores de recarga: Si la función de recarga $f(j)$ es relativamente chica y aumenta lentamente, es probable que el algoritmo deba iterar mas valores de la función ya que tardaría mas minutos en volverse mayor a las cantidad de enemigos actual. Si la función de recarga tiene valores altos o aumenta rápidamente, el algoritmo podría requerir menor tiempo para encontrar la mejor estrategia ya que con la función optimizar_funcion se dejarían de tomar en cuenta los valores de la función que sean demasiado grandes y entonces las iteraciones serían mas cortas.

3.2. Optimalidad

Si los valores de las llegadas de enemigos varían, es decir, si hay momentos con oleadas de enemigos grandes y otros con oleadas chicas, el algoritmo podría complicarse para encontrar una estrategia óptima debido a que no se podrían podar valores de la función. En oleadas chicas, el algoritmo podría llegar a podar algunos valores de la función dada por lo que funcionaría mas rápido, en cambio en oleadas grandes, probablemente se deban provar todos los valores de la funcion hasta n, siendo n el minuto de la batalla actual.

Variabilidad en los valores de recarga: La variabilidad en los valores de recarga también puede afectar la optimalidad del algoritmo. Como se menciono anteriormente si la funcion de recarga es, por ejemplo, exponencial o cuadrática luego de algunos minutos su valor sería muy grande y probablemente mayor al número de enemigos de cualquier oleada por lo tanto se podrían podar en la función optimizar_funcion por lo tanto para cada minuto se deberia iterar a través de menos valores de la función por lo tanto cada iteración seria mas corta, agilizando el algoritmo y probablemente haciendo que este funcione mas veloz que $\mathcal{O}(n^2)$. Por el contrario, si los valores de recarga son bajos, el algoritmo probablemente no pueda podar ningun valor en la función antes mencionada por lo tanto la complejidad sería como la del peor caso, es decir, $\mathcal{O}(n^2)$.

4. Pruebas

Para las pruebas se usaron los siguientes algoritmos:

Para generar las oleadas de enemigos y la funcion de cada batalla se utilizo el siguiente algoritmo:

```
def random_generator(max_enemigos, max_recarga, minutos):
    oleadas = []
    valoresFuncion = []
    for i in range(minutos):
        oleadas.append(random.randint(0, max_enemigos))
    valor_funcion = random.randint(0, max_recarga)
    for i in range(minutos):
        valoresFuncion.append(valor_funcion)
        valor_funcion += random.randint(0, max_recarga)
    return oleadas, valoresFuncion
```

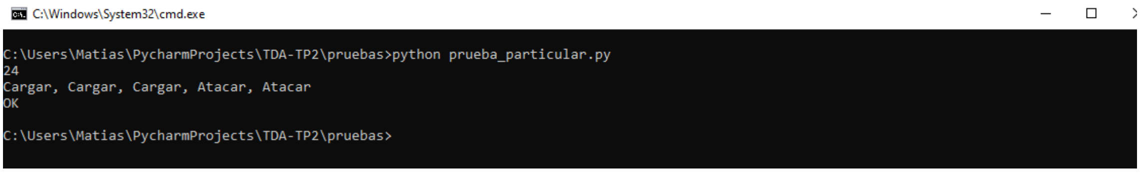
Ahora se resolvera un caso de una batalla de 5 minutos generada por el algoritmo presentado y se comparará la estrategia obtenida con la del algoritmo: La batalla en cuestion tiene las siguientes oleadas: 25, 8, 4, 23, 9 en este respectivo orden y los valores de la funcion dada son: 1, 10, 14, 23, 31. Notemos que si atacamos sin recargar solo eliminaremos a un enemigo en cualquiera de las batallas, por lo que nunca sera conveniente atacar sin recargar. Notemos que si atacamos antes del minuto 4 eliminaremos a lo sumo 19 enemigos, y si esperamos hasta el minuto 4 eliminaremos almenos 23 por lo que siempre es conveniente atacar recien en el minuto 4 y finalmente si atacamos recien en el minuto 4 en el minuto 5 no nos queda mas que atacar y eliminar a un enemigo mas por lo tanto el maximo numero de bajas que podemos realizar son: 24. Entonces, la estrategia a seguir sera: Cargar, Cargar, Cargar, Atacar, Atacar. A este caso lo guardaremos en el archivo caso1.txt y mediante el siguiente codigo veremos cual es la solucion dada por el algoritmo:

```
import pd

optimo, estrategia = pd.pd("caso1.txt")
print(optimo)
print(estrategia)
assert optimo == 24 and estrategia == "Cargar, Cargar, Cargar, Atacar, Atacar"
```

```python
print("OK")
```

Corriendo el código tenemos que:



5. Mediciones

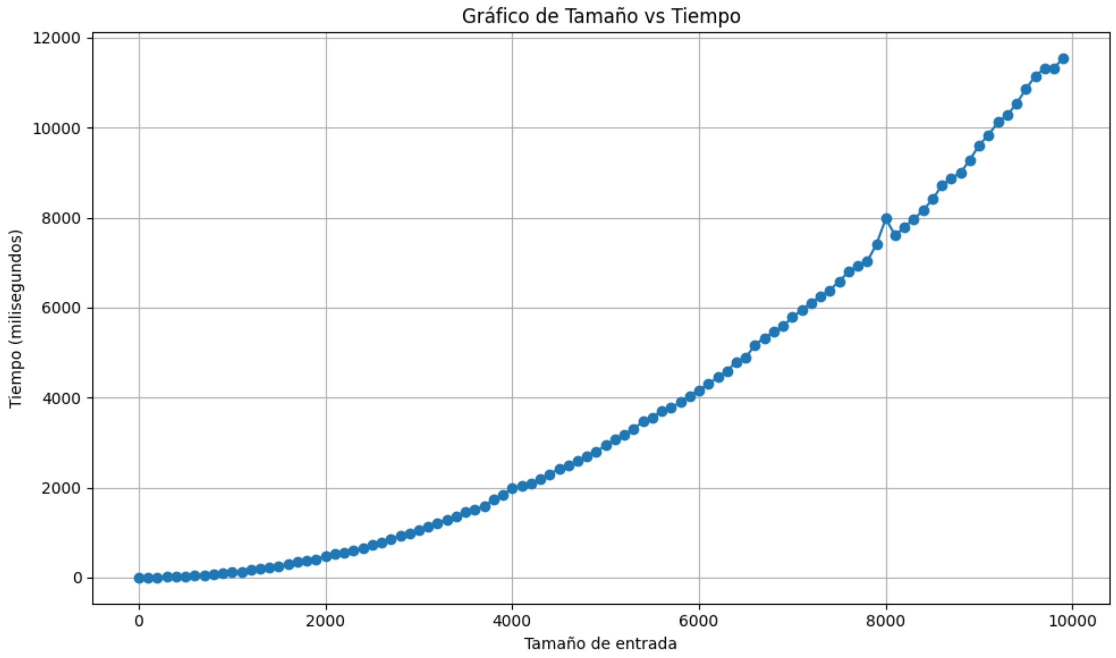
Se realizaron mediciones en base a resolver batallas de entre 1 a 100000 minutos de duracion, de 100 en 100 generadas por el random_generator.py y se resueltas con el algoritmo, midiendo el tiempo que tardaba cada en resolver cada batalla y luego, se hizo un grafico siendo x el tamaño de la entrada (los minutos que dura la batalla) e y los milisegundos que tardo el algoritmo en encontrar la mejor estrategia. Para aplicar el algoritmo de resolución de batallas y graficar lo mencionado anteriormente se utilizo el siguiente código: Para medir el tiempo que tardaba el algoritmo en encontrar la mejor estrategia para una batalla:

```python
def crear_tamanio_vs_tiempo(max_enemigos, max_recarga, minutos):
    oleadas, valoresFuncion = random_generator(max_enemigos, max_recarga, minutos)
    inicio = time.time()
    pd(minutos, oleadas, valoresFuncion)
    fin = time.time()
    duracion = fin - inicio
    return duracion * 1000
```

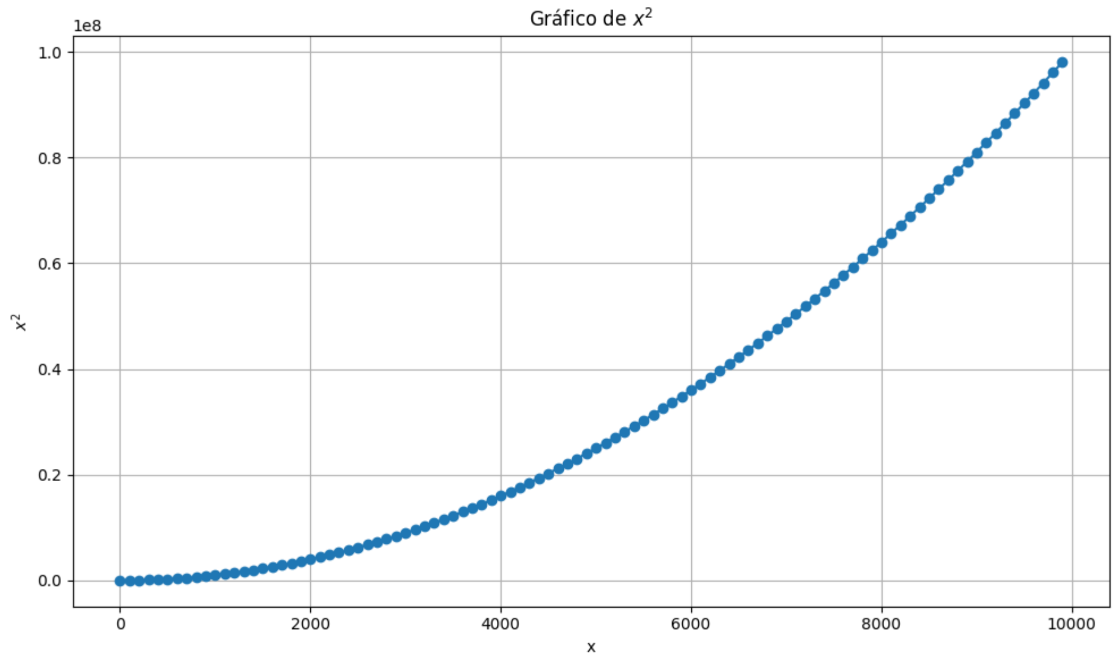
Para graficar la cantidad de elementos de una batalla y el tiempo tomado por el algoritmo en encontrar la estrategia además de los parametros pasados al generador de casos:

```python
MAX_ENEMIGOS = 5000
MAX_RECARGA = 10
TAMANIO_INICIAL = 1
TAMANIO_FINAL = 10000
TAMANIO_SALTO = 100
def graficarFuncionTamanio():
    listaTamanios = []
    listaDuraciones = []
    for i in range(TAMANIO_INICIAL, TAMANIO_FINAL, TAMANIO_SALTO):
        tiempo = crear_tamanio_vs_tiempo(MAX_ENEMIGOS, MAX_RECARGA, i)
        listaTamanios.append(i)
        listaDuraciones.append(tiempo)
    plt.figure(figsize=(10, 6))
    plt.plot(listaTamanios, listaDuraciones, marker='o', linestyle='-')
    plt.title('Grafico de Tamanio vs Tiempo')
    plt.xlabel('Tamanio de entrada')
    plt.ylabel('Tiempo (milisegundos)')
    plt.grid(True)
    plt.tight_layout()
    plt.show()
```

Y finalmente los resultado fueron:



Y a continuación el grafico de x^2 :



Como se puede apreciar, el algoritmo tienen una tendencia efectivamente cuadrática en función del tamaño de la entrada como se observa comparando la curva de ambas funciones.

6. Conclusiones

Después de haberse realizado y codificado el algoritmo por medio de la técnica de diseño "Programación Dinámica", con el objetivo de determinar cuando usar los ataques de fisuras para maximizar el número total de enemigos eliminados, se puede concluir que el resultado obtenido fue exitoso. El algoritmo pasa las pruebas propuestas por la cátedra de forma correcta y los valores de las cantidades de tropas eliminadas corresponden con los de los archivos de prueba proporcionados.

En cuanto al tiempo de ejecución, se determinó que el algoritmo tiene una complejidad computacional $O(n^2)$. Respecto a la variación de los valores de entrada, se concluyó que para valores grandes, el algoritmo tarda considerablemente más tiempo que cuando los tamaños de entrada son valores pequeños.