Trabajo Práctico 2: Programación Dinámica para el Reino de la Tierra

Facultad de Ingeniería de la Universidad de Buenos Aires

Teoría de Algoritmos

Cátedra Buchwald-Genender

Gómez Belis, Sofía
Padrón: 109358
email: sgomezb@fi.uba.ar

Llanos Pontaut, Valentina
Padrón: 104413
email: vllanos@fi.uba.ar

Orsi, Tomas Fabrizio
Padrón: 109735
email: torsi@fi.uba.ar

Indice

1 Análisis del problema

1.1 Descripción y objetivo

El objetivo de este trabajo práctico es implementar un algoritmo de programación dinámica para ayudar a los Dai Li, la policía secreta de la ciudad Ba Sing Se del Reino de la Tierra, en su combate contra la Nación del Fuego. Como somos sus jefes estratégicos, debemos reportarles las estrategias que deberían emplear en cada minuto del ataque con el fin de eliminar la máxima cantidad de maestros Fuego.

Afortunadamente, las mediciones sísmicas de los Dai Li nos permitieron obtener información de la cantidad de enemigos que llegan en cada minuto. Además, sabemos que la cantidad de enemigos no es acumulativa; después de cada ataque, los enemigos se retiran. Tenemos entonces, una lista de valores x_i que ilustran este hallazgo. Por otro lado, la potencia de los ataques de fisura de la policía secreta dependen de cuánto tiempo fue cargada la energía. La función monótona creciente $f(\cdot)$ indica que, si transcurrieron j minutos desde que se utilizó este ataque, entonces se podrá eliminar hasta $f(j)$ soldados enemigos. Como este valor puede ser mayor a la cantidad de maestros Fuego, en el minuto k se podrá eliminar $min(x_k, f(j))$ soldados, perdiendo además la energía acumulada. En el primer minuto, en caso de decidir atacar, les corresponde $f(1)$ de energía.

Sabiendo la duración en minutos del ataque (n), y los valores de x_i y $f(j)$, podemos analizar el problema presentado e informarles a los Dai Li la secuencia de estrategias que les permitirá eliminar la mayor cantidad de enemigos empleando ataques de fisuras.

1.2 Análisis y ecuación de recurrencia

La resolución de un problema por medio de la programación dinámica implica reutilizar las soluciones a subproblemas más pequeños en subproblemas más grandes que los incluyan. En el contexto actual, el foco está puesto en maximizar la cantidad de enemigos eliminados dados n minutos. Esta variable n es crucial en el análisis del problema planteado puesto que si se tiene k minutos, con $k < n$, necesariamente la cantidad de enemigos eliminados será menor o igual a la solución en el minuto n, siendo igual en el caso de que no lleguen más enemigos entre k y n minutos. De esta forma, la cantidad de minutos que tiene el ataque repercute en el resultado final del combate. Por ejemplo, si $n = 0$, se puede afirmar que la cantidad de enemigos eliminados será también 0. Entonces, nuestros subproblemas estarán dados por la cantidad máxima de enemigos que se pueden eliminar en cada minuto $i \leq n$.

Sabiendo la forma de los subproblemas, debemos analizar cómo se componen para resolver subproblemas más grandes. Si queremos obtener la solución óptima en el minuto i, vamos a poder utilizar las soluciones parciales calculadas hasta entonces, pero no nos interesa si existen o no problemas más grandes. En cada minuto $i < n$ hay solamente dos estrategias posibles: atacar o cargar.

Atacar implica enfrentarse con el enemigo, derrotando $f(j)$ soldados (con j siendo la cantidad de minutos desde el ataque anterior) y perdiendo la energía acumulada. En cambio, definimos **cargar** como la acción de decidir no atacar y acumular más energía para un ataque futuro.

Sin embargo, en el caso del minuto $i = n$, ¿tiene sentido cargar sabiendo que no van a llegar más enemigos en el futuro? No. Ésto se debe a que la cantidad de maestros Fuego que se podrá eliminar en ese minuto será mayor o igual a 0, pero si cargamos energía, será definitivamente nula. Por lo tanto, en el último minuto conviene siempre atacar. Ahora bien, como establecimos antes, en el minuto $i \leq n$ no importa si $i = n$ o $i < n$, solamente debemos calcular el óptimo actual. En base al análisis previamente presentado, siempre va a ser mejor atacar a cargar, más allá de que en la solución final (problema mayor) se lleve a cabo la estrategia opuesta debido a que eso esté contemplado en el óptimo del minuto n.

Si en el minuto i los Dai Li atacan, la cantidad de enemigos eliminados en ese instante será $min(f(j), x_i)$. El valor de x_i es conocido, pero la energía acumulada ($f(j)$) por la policía secreta de la ciudad depende de los minutos que pasaron desde el último ataque. De esta manera, tenemos una segunda variable involucrada en el problema: j. Si el último ataque fue realizado hace un minuto, actualmente se podrá eliminar $min(f(1), x_i)$ soldados y la cantidad de enemigos eliminados acumulada será la suma entre este valor y el correspondiente en el ataque anterior.

¿Qué sucede si el óptimo hace dos minutos es mayor que en el minuto anterior o su suma con $min(f(2), x_i)$ lo es? Como queremos maximizar el resultado final, claramente nos conviene haber atacado hace dos minutos, lo cual también indica que en el minuto $i-1$ se cargó energía. Tenemos varias opciones para el ataque anterior, más precisamente $1 \leq j \leq i$, pero utilizaremos aquel que nos lleve a la mejor solución. Entonces, el óptimo para el minuto i será la suma entre $min(f(j), x_i)$ y el óptimo en el minuto $i - j$.

Sabiendo la forma de los subproblemas y la manera en que éstos se combinan, podemos plantear la **ecuación de recurrencia** para el minuto i:

$$OPT[i] = max(min(f(j), x_i) + OPT[i - j] \forall j \in [1; i])$$

Como caso base, tenemos que en el minuto 0 se eliminan 0 enemigos.

Encontrada la ecuación de recurrencia, procedemos a aplicarla iterativamente de manera bottom up, construyendo las soluciones a los subproblemas de $i < n$ hasta llegar a la solución del problema original con $i = n$. Esta técnica es justamente programación dinámica. Empleamos `memoization` guardando los resultados calculados previamente en un arreglo. El procedimiento explicado nos permite realizar una exploración implícita del espacio de soluciones. La solución final será óptima porque en el minuto n elegimos haber atacado hace j minutos, donde j maximiza la ecuación de recurrencia.

Como conclusión, el uso de esta ecuación de recurrencia en nuestra implementación nos permite determinar la cantidad máxima de enemigos que se pueden atacar. En cada minuto calculamos el máximo número de adversar-

ios eliminados si se decide atacar en ese instante. Este valor está determinado por el tiempo entre éste y el ataque anterior. Al quedarnos con un j que maximice la suma final de enemigos derrotados en el minuto i, nos aseguramos de obtener una solución óptima a ese subproblema. Luego, en el último minuto del combate, utilizaremos las soluciones a estos subproblemas de forma tal que tendremos en cuenta el ataque anterior que ocurrió en el minuto $k = n - j$, donde $OPT[k] + min(f(j), x_n)$ maximiza la cantidad de enemigos eliminados en total. A su vez, el $OPT[k]$ tiene lo mismo en consideración.

Guardar los óptimos en un arreglo de soluciones parciales para cada minuto nos facilita reconstruir la estrategia de ataque óptima que permite obtener el resultado para $OPT[n]$. Sabiendo el valor de la cantidad de enemigos que llegan en el minuto n, x_n, podemos comparar resultados parciales con $OPT[n]$ para obtener j. Es decir, como ya determinamos que en el minuto n vamos a atacar, j va a corresponder al resultado que cumpla con $OPT(n) = OPT(k) + min(f(j), x_n)$, siendo k el minuto en que previamente se realizó un ataque ($k = n - j$). Entre k y n, la estrategia empleada es recargar fuerzas.

Repetimos el procedimiento para k, hasta llegar al minuto 0.

2 Complejidad algorítmica

2.1 Complejidad lectura de archivos

Antes de comenzar el algoritmo, tenemos que generar las listas de elementos sobre las cuales éste va a operar. Para esto tenemos la siguiente porción de código:

```
5  def generarTestDe(archivo):
6      x = []
7      f = []
8      with open(archivo, "r") as file:
9          n = None
10         elem_actual = 1
11         for i, line in enumerate(file):
12             line = line.strip()
13             if line.startswith("#"):
14                 continue
15             if not n:
16                 n = int(line)
17                 continue
18             if elem_actual <= n:
19                 x.append(int(line))
20                 elem_actual += 1
21             else:
22                 f.append(int(line))
23
```

```
24      return n, x, f
25
```

La función lee la línea que contiene la cantidad de valores que van a tener ambas listas. Una vez hecho eso, lee n líneas para almacenar los distintos valores de x_i y luego lee otras n líneas para tener los valores de $f(j)$. Esto tiene una complejidad temporal $O(2 \cdot n)$ que, despreciando la constante, resulta $O(n)$.

2.2 Complejidad algoritmo

```
10  def eliminar_enemigos_optimizado(n, x):
11      enemigos_eliminados = x
12      max_enemigos = sum(enemigos_eliminados)
13      secuencia = [ATACAR] * n
14      return (max_enemigos,secuencia)
15
```

El algoritmo empieza creando una lista de tamaño $n + 1$. Según la documentacion oficial , esto conlleva $\mathcal{O}(n)$ ya que se realiza n veces una operación de tiempo constante $\mathcal{O}(1)$.

Luego, en la línea `7` comienza un ciclo `for` que recorre todos los valores de n. En la línea `9` comienza un nuevo ciclo `for`, el cual va desde 0 hasta el minuto actual i. Esto significa que, en el peor de los casos, este ciclo va a iterar hasta n. Dentro del ciclo, todas las operaciones son $\mathcal{O}(1)$, es decir, constantes.

Al tener dos ciclos anidados que realiza la cantidad de iteraciones mencionadas, podemos afirmar que la complejidad algorítmica de las líneas `7` a `17` es $\mathcal{O}(n^2)$.

Luego, en la línea `20` se ejecuta la función `obtener_secuencia_estrategias`, la cual analizamos en Complejidad reconstrucción de la solución. Sin embargo, podemos adelantar que su complejidad será también cuadrática.

Finalmente, en la línea `21` se ejecuta el método `reverse()`, el cual tiene complejidad $\mathcal{O}(n)$

Es decir, que el algoritmo de programación dinámica tiene una complejidad

$$\mathcal{T}(n) = 2 \cdot \mathcal{O}(n^2) + \mathcal{O}(n) = \mathcal{O}(n^2)$$

2.3 Complejidad reconstrucción de la solución

```
36  def obtener_secuencia_estrategias(x, f, enemigos_eliminados, minuto_actual):
37      secuencia = []
38      while minuto_actual > 0:
39
40          secuencia.append(ATACAR)
41
42          for minutos_desde_ultimo_ataque in range(minuto_actual):
43
44              offsetMins = minuto_actual-minutos_desde_ultimo_ataque-1
```

```
45              enemigos_ataque_anterior = enemigos_eliminados[offsetMins]
46
47              cantReales = min(f[minutos_desde_ultimo_ataque], x[minuto_actual-1])
48              enemigos_actuales = cantReales
49
50              enemigosDerrotados = enemigos_ataque_anterior + enemigos_actuales
51              esIgual = enemigosDerrotados == enemigos_eliminados[minuto_actual]
52              if esIgual:
53                  minuto_actual = minuto_actual-minutos_desde_ultimo_ataque-1
54                  break
55              else:
56                  secuencia.append(CARGAR)
57
58      return secuencia
59
```

La reconstrucción de la secuencia estratégica empieza desde el último minuto, hasta 0. Vemos que en la línea `26` tenemos un ciclo `while`, el cual tiene dentro un ciclo `for`. Dentro del `for` todas las operaciones son de complejidad $\mathcal{O}(1)$.

En el peor de los casos, podemos pensar a estos dos ciclos como dos ciclos anidados que iteran desde 0 hasta `n`. Por esto, decimos que la reconstrucción va a tener una complejidad de $\mathcal{O}(n^2)$.

2.4 Efecto de las variables sobre el algoritmo

La complejidad obtenida es tenida en cuenta para el peor caso posible. Sin embargo, existe la posibilidad de que se ejecute en $O(n)$ si se recibe un set de datos con determinadas características.

Este es el caso cuando todos los datos de x_i son menores al resultado de $f[0]$ que se corresponde con un minuto de diferencia con el ataque anterior, es decir, $f(1)$. Esto se debe a que el resultado será predecible ya que se podría derrotar al 100% de enemigos en todos los ataques sin necesidad de cargar fuerzas.

La comprobación de si el conjunto de datos de entrada cumple esta condición es realizada antes de ejecutar el algoritmo propiamente dicho. Es decir, primero utilizamos la función `esOptimizable(n, x, f)` que hace la verificación en tiempo lineal. Si efectivamente cumple el requisito, `eliminar_enemigos_optimizado(n, x)` resuelve el problema planteado en tiempo $O(n)$ sumando la totalidad de enemigos y estableciendo una secuencia en la que siempre se ataca. Si bien este algoritmo no es de programación dinámica, creemos que es fundamental tenerlo en cuenta especialmente con sets de datos muy grandes donde la complejidad lineal por sobre la cuadrática puede hacer una gran diferencia en el timepo de ejecución. Por el contrario, si no se puede optimizar, entonces ejecutamos nuestro algoritmo de programación dinámica. En este caso realizar la comprobación no afecta la complejidad resultante, sigue siendo $O(n^2)$.

```
4   def esOptimizable(n, x, f):
5       for minuto_actual in range(n):
```

```
6           if x[minuto_actual] > f[0]:
7               return False
8       return True
9
10  def eliminar_enemigos_optimizado(n, x):
11      enemigos_eliminados = x
12      max_enemigos = sum(enemigos_eliminados)
13      secuencia = [ATACAR] * n
14      return (max_enemigos,secuencia)
15
```

3 Ejemplos de ejecución

Se pueden encontrar en la carpeta `ejemplos_adicionales` del repositorio entregado algunos ejemplos de posibles ejecuciones creadas para corroborar la optimalidad del algoritmo planteado.

Además de los ejemplos brindados por la cátedra, se pensó en distintas perspectivas que podrían *comprometer* de alguna manera la solución del algoritmo. Algunos aspectos evaluados fueron:

- Momento en el que realiza el primer ataque:
  - `atacar_al_inicio`: El primer ataque es al inicio del combate
  - `atacar_a_la_mitad`: El primer ataque es a la mitad del combate
  - `atacar_al_final`: El primer ataque es al final del combate
  - `atacar_siempre`: No conviene nunca reservar energía, ataca siempre
- Relación numérica entre x_i y $f(j)$
  - `x_menor_a_f`: Todos los x_i son menores a todos los $f(j)$
  - `x_mayor_a_f`: Todos los x_i son mayores a todos los $f(j)$
  - `x_igual_a_f`: Todos los x_i sin iguales a todos los $f(j)$ (con $i == j$)
  - `igual_xi_menor_a_f0`: Todos los x_i son iguales entre sí y menores a f(0)
  - `igual_xi_mayor_a_f0`: Todos los x_i son iguales entre sí y mayores a f(0)
- Diferencia numérica entre x_i y x_{i+1}, y $f(j)$ y $f(j+1)$
  - `dif_x_cero`: La diferencia numérica entre cada x_i es nula
  - `dif_x_chica`: La diferencia numérica entre cada x_i es de 10^{-i}
  - `dif_x_grande`: La diferencia numérica entre cada x_i es de 10^{i}
  - `dif_f_chica`: La diferencia numérica entre cada $f(j)$ es de 10^{-j}

  - `dif_f_grande`: La diferencia numérica entre cada $f(j)$ es de 10^j

- Ordenamiento de x_i
  - `xi_descendente`: Los x_i están ordenados de manera descendente
  - `xi_ascendente`: Los x_i están ordenados de manera ascendente
    Observación: Los valores de $f(j)$ siempre estarán ordenados de manera ascendente por ser una función monótona creciente

- Volumen de ataques recibidos
  - `x_menor_a_f_volumen`: Todos los x_i son menores a $f(0)$ pero con una muestra de 100 combates

Los resultados de los ejemplos se podrán ejecutar en la terminal con los siguientes comandos:

- `python3 codigo/main.py`: Imprime todos los resultados de ejemplos adicionales y ejemplos de cátedra con su valor obtenido y esperado, incluyendo el tiempo de ejecución
- `python3 codigo/main.py ejemplos_adicionales/atacar_al_inicio.txt` : Si se quiere imprimir los resultados de un ejemplo particular
- `python3 codigo/main.py ejemplos_adicionales/atacar_al_inicio.txt --mostrarSecuencia` : Si se quiere imprimir los resultados de un ejemplo particular mostrando además la secuencia de estrategias implementadas

4 Análisis de optimalidad ante la variabilidad de x_i y $f(j)$

Como mencionamos anteriormente, nuestra implementación hace una exploración implícita de todo el espacio de soluciones. Esto se ve reflejado tanto en el código como en la ecuación de recurrencia. Para buscar el par de valores que maximice la cantidad de enemigos derrotados, iteramos por todos los valores posibles de $f(j)$ dado un x_i.

Esto implica que no hay valor de x_i y $f(j)$ que no sea explorado por el algoritmo. Es decir, mientras que se cumplan las condiciones iniciales del problema (haya la misma cantidad de valores para x_i y $f(j)$ y además que $f(j)$ sea monótona creciente), el algoritmo va a llegar a la solución optima.

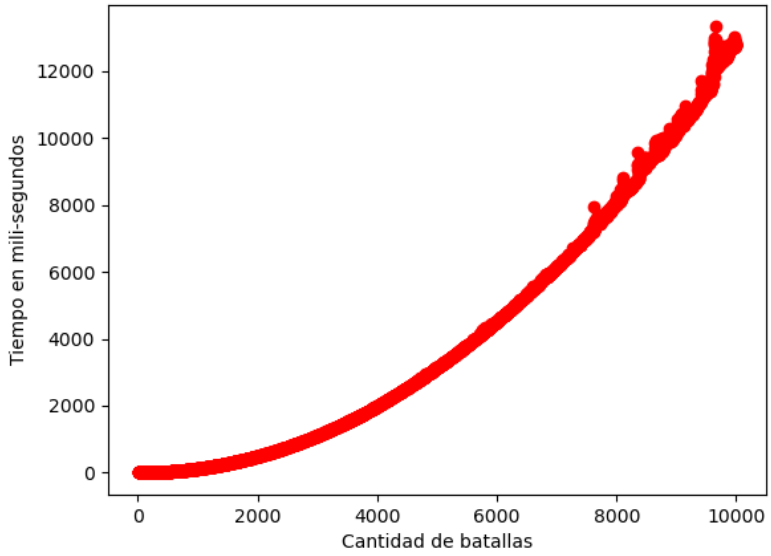
5 Mediciones de tiempo

Para corroborar la complejidad algorítmica de nuestra implementación, realizamos una serie de tests de volumen. Éstos tienen en cuenta el algoritmo de programación dinámica, sin la optimización explicada en Efecto de las variables

sobre el algoritmo. A diferencia del TP1 y debido a la complejidad temporal del algoritmo, realizamos tests con una menor cantidad de casos. Para poder graficar y analizar el comportamiento de la implementación, realizamos dos corridas, una con alrededor de 3500 tests con un n entre 1 y 10000; y otra con alrededor 500 tests 500 tests, con n entre 0 y 5000. El código que usamos para generar los tests se encuentran en el archivo `codigo/grafico_complejidad.py`. Empleamos la biblioteca random para generar valores aleatorios de x y f.

Observando los gráficos podemos entender mejor el comportamiento de nuestra implementación.

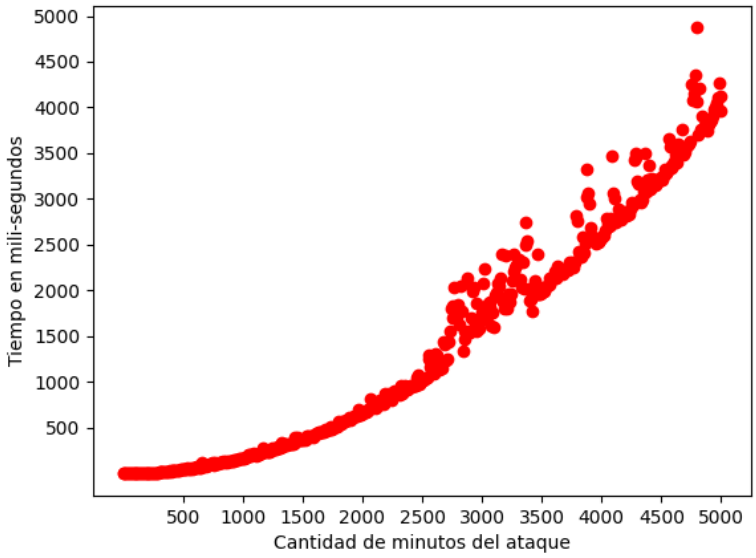
Con la primera corrida de tests obtuvimos:



Podemos ver que el gráfico tiene una forma cuasi parabólica, lo cual refleja el comportamiento cuadrático del algoritmo.

Además, podemos ver el rápido crecimiento que toman los valores. Por ejemplo, 2000 minutos de combate tardan alrededor de 500 mili-segundos y con 4000 el algoritmo tarda aproximadamente 2000 mili-segundos. Vemos que al duplicar el tamaño de la entrada, el tiempo de ejecución no se duplica, sino que se cuadruplica.

La segunda corrida de tests tuvo resultados análogos:



Al igual que el otro gráfico, este también tiene una forma parabólica. Podemos observar que con 1000 minutos de combate tarda alrededor de 160 mili-segundos y con $n = 2000$ conlleva más de 600 mili-segundos.

Para comparar con el gráfico anterior, calculamos el tiempo de ejecución exacto para $n = 2000$ y $n = 4000$, donde n es la cantidada de minutos en la que llegan enemigos de la Nación del Fuelgo. Obtuvimos 658 y 2567 mili-segundos, respectivamente, verificando que el patrón establecido se repite.

Con esto concluimos que, a medida que aumenta el tamaño de entrada n (cantidad de minutos), el tiempo de ejecución del algoritmo crece a un ritmo cuadrático. Esto coincide con el análisis de complejidad previamente presentado, indicando un costo proporcional a n^2.