

TEORÍA DE ALGORITMOS  
(75.29) CURSO BUCHWALD - GENENDER

# Trabajo Práctico 1

## Algoritmos Greedy en la Nación del Fuego

2 de mayo de 2024

Thiago Pacheco  
111003

Matias Bartellone  
110484

Iñaki Llorens  
107552

## 1. Objetivos del trabajo

La idea del trabajo práctico es consolidar los conocimientos adquiridos sobre el funcionamiento y el diseño de Programación Dinámica, para esto se deberá analizar el problema planteado y proponer un algoritmo de Programación Dinámica que encuentre la solución óptima al problema.

## 2. Análisis del problema

El problema a resolver planteado por la cátedra trata sobre cómo plantear la defensa de una ciudad en medio de una guerra. Cómo líderes estratégicos del Reino de la Tierra, tenemos que intentar eliminar a la mayor cantidad de soldados enemigos, que llegan en ráfagas a atacarnos. Para coordinar la defensa de la ciudad tenemos que tener en cuenta que nuestros soldados, el Dai Li, tienen un nivel de energía que representa cuantas tropas enemigas pueden eliminar. Una vez que nuestros soldados atacan, gastan toda su energía y deberán esperar unos minutos para poder recuperarse y volver a atacar.

Debemos ayudar a los Dai Li a determinar en que minutos es conveniente atacar para obtener la mayor cantidad de bajas posibles, donde las variables son:

- $x_n$ : cantidad de soldados que llegaron en el minuto  $n$ -ésimo.
- $j$ : minutos sin atacar acumulados.
- $n$ : minutos de la pelea acumulados
- $f(j)$ : cantidad de posibles enemigos abatidos con  $j$  minutos acumulados.

Teniendo estos datos que se reciben al inicio del problema, queda en nosotros analizarlos para poder encontrar un algoritmo que encuentre siempre la solución óptima, es decir lo que debemos determinar es en que minutos cargar energía de los soldados o atacar para eliminar tantos enemigos como sea posible.

Para maximizar la eliminación de enemigos, teniendo en cuenta que los soldados deben recargar su energía tenemos que tener en cuenta que:

- El minuto en que se descansa no se genera una ganancia.
- Si atacamos en un minuto  $i$ , se gasta toda la energía y al minuto siguiente se tiene  $j = 1$  de minutos sin atacar acumulados.

Al analizar en profundidad el problema se puede ver que hay dos dimensiones que afectan a cómo se componen los subproblemas, estos son  $n$ , minutos de la pelea acumulados, y  $j$ , minutos sin atacar acumulados. Si quisiésemos pensar el problema solamente teniendo en cuenta como va avanzando  $n$ , que capaz es lo que uno pensaría inicialmente ya que en Programación Dinámica es muy común que los problemas se compongan de esta manera, uno no estaría analizando todas las soluciones posibles porque algo que es importante de resaltar es que se tiene que pensar en cada minuto cómo nos convenía haber llegado. Veamos un paso a paso de lo que tuvimos en cuenta inicialmente para llegar a la solución:

1. Para 1 minuto: siempre voy a atacar con energía acumulada  $j = 1$  y las bajas enemigas son  $\min(x_1, f(1))$ , y este es el caso base.
2. Para 2 minutos: para el segundo minuto podemos llegar:
  - a) Habiendo atacando el minuto 1
  - b) Habiendo descansado el minuto 1

Y habría que calcular cual es el máximo entre atacar con  $j = 1$  los minutos 1 y 2, o el de descansar el minuto 1 y atacar con  $j = 2$  en el minuto 2.

3. Para 3 minutos: para el tercer minuto podemos llegar:

- a) Habiendo atacando el minuto 1 y atacado el minuto 2
- b) Habiendo descansado el minuto 1 y atacado el minuto 2
- c) Habiendo descansado el minuto 1 y descansado el minuto 2

Y acá habría que calcular cual es el máximo entre atacar con  $j = 1$  los minutos 1 y 2 y 3, o el de descansar el minuto 1 y atacar con  $j = 2$  en el minuto 2 y luego atacar con  $j = 1$  en el minuto 3, o o el de descansar en los minutos 1 y 2 y atacar con  $j = 3$  en el minuto 3.

Y así se debería seguir enumerando hasta el minuto  $n$ ...

Vemos acá que en un minuto  $j$  podemos haber llegado habiendo hecho varias combinaciones diferentes de cargar o atacar, y esto pasa porque nuestro problema crece a medida que lo hace  $n$  y la ganancia de cada minuto depende de como se llegó al mismo. Por lo tanto si nos paramos en un minuto  $i$  y teniendo ya calculado de antes cuál es la mejor combinación de cargas o ataques podemos usar esa información para ver que es lo mejor que se puede hacer en ese día  $i$ . Pero esto solo se logra si nos guardamos las combinaciones de cargas y ataques.

Dado el problema, que requiere la implementación de programación dinamica para minimizar la repetición de sub-calculos optimos necesarios para operar, se ha determinado que una estrategia eficaz consiste en calcular el maximo entre la cantidad de bajas posibles atacando cada minuto más la suma acumulada de los minutos restantes.

### 3. Propuesta de Solución

Para llegar a la solución óptima, hicimos un analisis del problema y buscamos cual seria el subproblema en cada caso para llegar a la resolucion del objetivo final. Llegamos a la conclusion de que podemos plasmar el problema en una matriz de  $n \times j$  donde  $n$  representa los minutos pasados en la batalla y  $j$  los distintos niveles de carga a los que se podria llegar cada minuto. La idea del algoritmo es ir recorriendo esa matriz por cada minuto  $n$  y en el minuto  $n$  ir recorriendo los distintos niveles de carga. Entonces, en el minuto  $n_i$  con un nivel de carga  $j_i$  tomamos el maximo de dos opciones:

- Podemos decidir no atacar ese día  $n$  con ese nivel de carga  $j$ , por lo que ese día usaria el optimo del mismo minuto pero con un nivel de carga - 1
- Podemos decidir atacar ese minuto  $n$  con ese nivel de carga  $j$ , por lo que el optimo seria la suma de el optimo en el minuto  $n-j$  con carga  $n-j$  y la cantidad de soldados derribados con el nivel de carga actual

Esto se puede ver representado en la ecuacion de recurrencia:

$$OPT(n, j) = \max \begin{cases} OPT(n, j - 1) \\ OPT(n - j, n - j) + \min(f(j), x_n) \end{cases}$$

De esta manera vamos guardando el optimo  $n_i, j_i$  en la matriz hasta llegar al final, donde va a quedar el valor maximo de soldados derribados de la forma optima.

Para la reconstruccion de la solucion, tomamos la matriz de optimos calculada previamente y aplicamos la ecuacion de recurrencia para guardar los días en los que se ataco. Si el valor actual es distinto al valor de una carga  $j$  anterior significa que ese día atacamos y seguiriamos recorriendo desde el valor  $n - jn - j$ . Si no atacamos seguimos buscando con valores de carga  $j$  menores.

## 4. Algoritmo

El algoritmo es simplemente un for anidado que recorre la matriz  $n \times j$  iterativamente para almacenar el valor optimo segun la ecuacion de recurrencia

```

1 def defensa_optima(x,f,n):
2     if n == 0:
3         return 0
4
5     OPT = [[0] * (n+1) for _ in range(n+1)]
6
7     for m in range(1, n+1):
8         for j in range(1, m+1):
9             OPT[m][j] = max(OPT[m][j-1], (OPT[m-j][m-j] + min(f[j], x[m])))
10
11     return OPT

```

Luego tenemos la reconstruccion del resultado, que como fue explicado anteriormente, recorre la matriz inversamente al algoritmo hasta encontrar un cambio en los valores y salta al día n-j que representa el día anterior en el que se ataco.

```

1 def reconstruccion(OPT, n):
2     reco = []
3     max_bajas = OPT[n][n]
4     m, j = n,n
5     while j != 0:
6         if OPT[m][j] != OPT[m][j-1]:
7             reco.insert(0, "ATACAR")
8             for i in range(j-1):
9                 reco.insert(0, "DESCANSAR")
10            m -= j
11            j = m
12        else:
13            j -= 1
14
15    return reco, max_bajas

```

### 4.1. Seguimiento

En las siguientes fotos mostramos un seguimiento de la matriz del algoritmo con el ejemplo de 5 elementos dado por la cátedra. Las celdas amarillas representan los minutos  $n$ , las naranjas las cargas  $j$  y las grises los valores base o no considerados (que valen 0). También se muestra a la derecha una columna  $x_i$  con los valores de los soldados atacantes en cada minuto y una columna  $f_i$  con los valores de carga en cada minuto cargado.

| n \ j | 0 | 1 | 2 | 3 | 4 | 5 | $x_i$ | $f_i$ |
|-------|---|---|---|---|---|---|-------|-------|
| 0     | 0 | 0 | 0 | 0 | 0 | 0 | 271   | 21    |
| 1     | 0 |   | 0 | 0 | 0 | 0 | 533   | 671   |
| 2     | 0 |   |   | 0 | 0 | 0 | 916   | 749   |
| 3     | 0 |   |   |   | 0 | 0 | 656   | 833   |
| 4     | 0 |   |   |   |   | 0 | 664   | 1543  |
| 5     | 0 |   |   |   |   |   |       |       |

Figura 1: Completo casos base

| n \ j | 0 | 1  | 2 | 3 | 4 | 5 | $x_i$ | $f_i$ |
|-------|---|----|---|---|---|---|-------|-------|
| 0     | 0 | 0  | 0 | 0 | 0 | 0 | 271   | 21    |
| 1     | 0 | 21 | 0 | 0 | 0 | 0 | 533   | 671   |
| 2     | 0 |    |   | 0 | 0 | 0 | 916   | 749   |
| 3     | 0 |    |   |   | 0 | 0 | 656   | 833   |
| 4     | 0 |    |   |   |   | 0 | 664   | 1543  |
| 5     | 0 |    |   |   |   |   |       |       |

Figura 2: Paso 1

Empezamos con el algoritmo completando los casos base y los no considerados con 0s, luego seguimos viendo cual sería lo optimo para hacer en minuto 1, y como es el caso base siempre deberíamos atacar, entonces vemos el mínimo entre  $x_1$  y  $f_1$ , y da que la cantidad de soldados derrotados es 21. Luego seguimos para analizar el minuto 2.

| n \ j | 0 | 1  | 2 | 3 | 4 | 5 | xi  | fi   |
|-------|---|----|---|---|---|---|-----|------|
| 0     | 0 | 0  | 0 | 0 | 0 | 0 | 271 | 21   |
| 1     | 0 | 21 | 0 | 0 | 0 | 0 | 533 | 671  |
| 2     | 0 | 42 |   | 0 | 0 | 0 | 916 | 749  |
| 3     | 0 |    |   |   | 0 | 0 | 656 | 833  |
| 4     | 0 |    |   |   |   | 0 | 664 | 1543 |
| 5     | 0 |    |   |   |   |   |     |      |

Figura 3: Paso 2

| n \ j | 0 | 1  | 2   | 3 | 4 | 5 | xi  | fi   |
|-------|---|----|-----|---|---|---|-----|------|
| 0     | 0 | 0  | 0   | 0 | 0 | 0 | 271 | 21   |
| 1     | 0 | 21 | 0   | 0 | 0 | 0 | 533 | 671  |
| 2     | 0 | 42 | 533 | 0 | 0 | 0 | 916 | 749  |
| 3     | 0 |    |     |   | 0 | 0 | 656 | 833  |
| 4     | 0 |    |     |   |   | 0 | 664 | 1543 |
| 5     | 0 |    |     |   |   |   |     |      |

Figura 4: Paso 3

La fila del 2 representa  $n = 2$ , como se puede ver en la matriz tenemos dos casos para analizar acá. Estos son llegar al 2 con 1 de energía acumulada o llegar al 2 con 2 de energía acumulada, vemos que pasa en cada caso. Si llegamos habiendo atacado en el minuto anterior vamos a tener que atacar con el mínimo de energía, que resulta en eliminar 21 enemigos de nuevo, y lo sumamos con lo que tenemos en el óptimo de  $n = 1$  y da 42. Por otro lado, si hubiesemos elegido cargar en el minuto 1, nos queda nada más atacar con 2 de energía acumulada y esto resulta en 533 bajas. Queda cómo óptimo y pasamos al minuto 3.

| n \ j | 0 | 1   | 2   | 3 | 4 | 5 | xi  | fi   |
|-------|---|-----|-----|---|---|---|-----|------|
| 0     | 0 | 0   | 0   | 0 | 0 | 0 | 271 | 21   |
| 1     | 0 | 21  | 0   | 0 | 0 | 0 | 533 | 671  |
| 2     | 0 | 42  | 533 | 0 | 0 | 0 | 916 | 749  |
| 3     | 0 | 554 |     |   | 0 | 0 | 656 | 833  |
| 4     | 0 |     |     |   |   | 0 | 664 | 1543 |
| 5     | 0 |     |     |   |   |   |     |      |

Figura 5: Paso 4

| n \ j | 0 | 1   | 2   | 3 | 4 | 5 | xi  | fi   |
|-------|---|-----|-----|---|---|---|-----|------|
| 0     | 0 | 0   | 0   | 0 | 0 | 0 | 271 | 21   |
| 1     | 0 | 21  | 0   | 0 | 0 | 0 | 533 | 671  |
| 2     | 0 | 42  | 533 | 0 | 0 | 0 | 916 | 749  |
| 3     | 0 | 554 | 692 |   | 0 | 0 | 656 | 833  |
| 4     | 0 |     |     |   |   | 0 | 664 | 1543 |
| 5     | 0 |     |     |   |   |   |     |      |

Figura 6: Paso 5

| n \ j | 0 | 1   | 2   | 3   | 4 | 5 | xi  | fi   |
|-------|---|-----|-----|-----|---|---|-----|------|
| 0     | 0 | 0   | 0   | 0   | 0 | 0 | 271 | 21   |
| 1     | 0 | 21  | 0   | 0   | 0 | 0 | 533 | 671  |
| 2     | 0 | 42  | 533 | 0   | 0 | 0 | 916 | 749  |
| 3     | 0 | 554 | 692 | 749 |   | 0 | 656 | 833  |
| 4     | 0 |     |     |     |   | 0 | 664 | 1543 |
| 5     | 0 |     |     |     |   |   |     |      |

Figura 7: Paso 6

Ahora es cuando el algoritmo se empieza a poner más complicado porque la cantidad de combinaciones posibles empieza a aumentar mucho. En  $n = 3$  y  $j = 1$  tomamos el óptimo de la fila anterior y lo sumamos con 21. En  $n = 3$  y  $j = 2$  vemos que el óptimo aumenta porque los enemigos que se pueden eliminar con energía acumulada 2 son 671 y en  $x_3$  llegan 916 tropas por lo que se usa la energía al máximo. Igualmente el óptimo de esta fila llega con  $n = 3$  y  $j = 3$  porque los enemigos que se pueden eliminar son 749. Este queda como óptimo por ahora.

| n \ j | 0 | 1   | 2    | 3    | 4    | 5 | xi  | fi   |
|-------|---|-----|------|------|------|---|-----|------|
| 0     | 0 | 0   | 0    | 0    | 0    | 0 | 271 | 21   |
| 1     | 0 | 21  | 0    | 0    | 0    | 0 | 533 | 671  |
| 2     | 0 | 42  | 533  | 0    | 0    | 0 | 916 | 749  |
| 3     | 0 | 554 | 692  | 749  |      | 0 | 656 | 833  |
| 4     | 0 | 770 | 1189 | 1189 | 1189 | 0 | 664 | 1543 |
| 5     | 0 |     |      |      |      |   |     |      |

Figura 8: Minuto 4 completo

En el minuto 4 podemos ver que llegamos al óptimo cuando la energía acumulada es 2, 1189 sale de la suma de 533 y 656, esto significa que por ahora el óptimo es cargar en el minuto 1, atacar en el 2, cargar en el 3 y atacar en el 4, matando así a 656 enemigos en este minuto que es el mínimo entre 656 y 671. Cómo en los minutos siguientes el óptimo da más bajo, nos quedamos con 1189.

| n \ j | 0 | 1    | 2    | 3    | 4    | 5    | xi  | fi   |
|-------|---|------|------|------|------|------|-----|------|
| 0     | 0 | 0    | 0    | 0    | 0    | 0    | 271 | 21   |
| 1     | 0 | 21   | 0    | 0    | 0    | 0    | 533 | 671  |
| 2     | 0 | 42   | 533  | 0    | 0    | 0    | 916 | 749  |
| 3     | 0 | 554  | 692  | 749  | 0    | 0    | 656 | 833  |
| 4     | 0 | 770  | 1189 | 1189 | 1189 | 0    | 664 | 1543 |
| 5     | 0 | 1210 | 1413 | 1413 | 1413 | 1413 |     |      |

Figura 9: Minuto 5 completo

Para terminar con el seguimiento, vemos que finalmente se ha llegado al óptimo y podemos hacer la reconstrucción en base a los datos que fuimos obteniendo, si miramos la solución desde el final hasta el principio. En la fila del minuto 5, los ultimos 4 valores son iguales y maximos en toda la matriz. Esto quiere decir que se ataco en el minuto 5 con una fuerza de carga 2, ya que se ve que el valor (5, 2) cambió del valor (5, 1). Si ataco con fuerza 2 significa que el anterior ataque fue en el minuto 3 con fuerza 3, por lo que el anterior seria en el 0,0 que no representa nada. Es decir, se ataco en los minutos 3 y 5. Y la solución final sería: [ CARGAR, CARGAR, ATACAR, CARGAR, ATACAR ].

## 4.2. Complejidad

Para el calculo de complejidad decimos que al recorrer toda la matriz  $n \times j$  tendríamos una complejidad de  $\mathcal{O}(n \cdot j)$ , pero como  $j$  esta capeado por  $n$  podemos simplificarlo a  $\mathcal{O}(n^2)$

## 4.3. Optimalidad

La optimalidad no es afectada por la variabilidad de los valores ya que para cada celda en la matriz calculamos el optimo hasta ese punto sin importar los diferentes valores que pueda llegar a haber.

## 5. Mediciones temporales

Se llevaron a cabo mediciones del rendimiento del algoritmo de optimización de defensas utilizando las pruebas de volumen proporcionadas por la cátedra, además agregamos más pruebas con valores que generamos aleatoriamente (y que no cambian porque son aquellos que guardamos en el repo), estas son pruebas de 1500, 2000, 2500, 3000, 3500, 4000 y 4500 elementos.

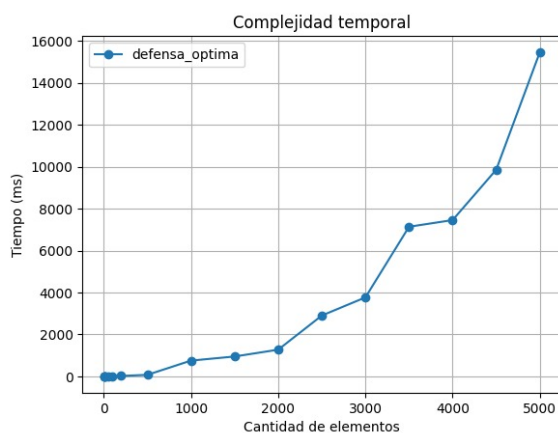


Figura 10: Medición temporal del algoritmo de Defensa Óptima



Figura 11: Comportamiento de un algoritmo  $\mathcal{O}(n^2)$

Ploteamos los resultados de las mediciones en un gráfico para mostrar como varían los tiempos de ejecución según la cantidad de elementos. Como se puede apreciar, el algoritmo Optimizar Batallas tiene una tendencia que efectivamente es igual a la de un algoritmo  $O(n^2)$  en función del tamaño de la entrada.

## 6. Conclusiones

En conclusión, podemos afirmar que el algoritmo en si es simple y facil de leer como muchos algoritmos de programacion dinamica, dado que se basa en hacer un recorrido iterativo (bottom up). Sin embargo, encontrar la ecuacion de recurrencia es la parte desafiante del problema, ya que representa como resolver la optimalidad en cada subproblema. Una vez resulta, resulta sencillo pasarlo a codigo.