Teoría de Algoritmos

Curso Buchwald - Genender

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

6 de mayo de 2024

Candela Matelica	110641
Jesabel Pugliese	110860
Yamila Shih	110521

1. Introducción

El presente informe reúne la documentación para el segundo trabajo práctico de la materia Teoría de Algoritmos, cuyo objetivo es resolver un problema utilizando la metodología de resolución de Programación Dinámica.

2. Presentación del problema

Ba Sing Se es una gran ciudad del Reino de la Tierra que ha logrado soportar los ataques de la Nación del Fuego. Los Dai Li la defienden utilizando técnicas de artes marciales, Tierra-control, y algunos algoritmos. Nosotros somos los jefes estratégicos de los Dai Li. Gracias a las técnicas de Tierra-control, lograron detectar que la Nación del Fuego planea un ataque ráfaga con miles de soldados maestros Fuego. El ataque sería de la siguiente forma:

- Ráfagas de soldados llegarían durante el transcurso de n minutos.En el i-ésimo minuto llegarán x soldados. Gracias a las mediciones sísmicas hechas con sus técnicas, los Dai Li lograron obtener los valores de $x_1, x_2, .., x_i$.
- Cuando los integrantes del equipo juntan sus fuerzas, pueden generar fisuras que permiten destruir parte de las armadas enemigas. La fuerza de este ataque depende cuánto tiempo se utilizó para cargar energía. Más específicamente, podemos decir que hay una función $f(.)$ que indica que si transcurrieron j minutos desde que se utilizó este ataque, entonces es capaz de eliminar hasta $f(j)$ soldados enemigos.
- Si se utiliza este ataque en el k-ésimo minuto, y transcurrieron j minutos desde su último uso, entonces se eliminará a mín$(x_k, f(j))$ soldados (y luego de su uso, se utilizó toda la energía que se había acumulado).
- Inicialmente los Dai Li comienzan sin energía acumulada (es decir, para el primer minuto, le correspondería $f(1)$ de energía si decidieran atacar inmediatamente).
- La función de recarga será una función monótona creciente.

3. Consigna

Se propone hacer un análisis del problema, planteando la ecuación de recurrencia correspondiente y proponiendo un algoritmo por programación dinámica que obtenga la solución óptima al problema planteado.

El problema consiste en maximizar la cantidad de enemigos eliminados por los Dai Li en Ba Sing Se mediante ataques de fisuras estratégicos, dados una secuencia de llegadas de enemigos y una función de recarga de energía. Se propone un algoritmo de programación dinámica para calcular eficientemente la cantidad máxima de enemigos eliminados y los momentos óptimos para los ataques. La variabilidad en los valores de llegadas de enemigos y recargas puede influir en los tiempos de ejecución y en la efectividad del algoritmo, aunque se espera que este sea óptimo en términos de maximización de eliminaciones.

4. Resolución

4.1. Análisis del problema

Primero, definamos las variables que aparecen en nuestro problema:

- $n \rightarrow$ minutos totales a considerar

- $x_1, x_2, \ldots, x_n \rightarrow$ llegada de enemigos por minuto
- $f(\cdot) \rightarrow$ función de recarga
- $i \rightarrow$ minuto actual
- $j \rightarrow$ minutos transcurridos desde la última vez que se atacó

Para el análisis de nuestro problema, comenzamos plantéandonos qué sucedería en un minuto dado i, donde tenemos una dicotomía entre atacar o recargar. Dada una llegada de enemigos x_i y un valor de recarga $f(j)$, sabíamos que al atacar, j debería reiniciarse en 1, y que al recargar se debería incrementar.

Nuestra primer idea fue directamente buscar cuál de las dos opciones maximizaba la cantidad de enemigos eliminados, pero fue descartada inmediatamente cuando nos dimos cuenta que esta opción daría siempre *Atacar*, pues al cargar ningún enemigo se era eliminado.

Esto nos llevó a pensar en que no alcanzaría con únicamente analizar la situación actual, sino que tendríamos que analizar qué sucede en los minutos anteriores y cómo esto afectaría a nuestro resultado actual. En cierto minuto i debíamos considerar no solo la opción *Atacar* o *Cargar*, sino también considerar lo que pasaría si atacamos con antes haber cargado.

Con esto, nuestra idea fue encaminada para el lado de ver también lo que pasaría si atacamos con antes haber cargado (en el minuto anterior $i-1$), y como la opción de *Cargar* ya estaría comprendida en el siguiente minuto $i+1$, nuestras opciones para el minuto i se redujeron a, o bien atacar (mín$(x_i, f(j_{actual}))$), o bien atacar con antes haber cargado (mín$(x_i, f(j_{anterior}+1))$), y quedarnos con el máximo.

Pero este enfoque tampoco nos iba a dar la solución óptima, pues en cada minuto estaríamos viendo tan solo dos opciones de ataque: para $j = 0$ y para $j = j_{anterior} + 1$, y no estaríamos considerando las opciones de atacar teniendo en cuenta otras recargas.

Así llegamos a que el problema no era tan solo pensar en atacar o recargar para cierto minuto i, sino que, de cierta forma, tendríamos que comparar las opciones que supondrían atacar y recargar para todos los valores posibles de j. En este momento, nos planteamos las alternativas que teníamos para almacenar dichos óptimos y elegimos que utilizar listas sería lo más adecuado para reducir la complejidad espacial, aunque iba a ser lo mismo almacenarlo en tablas o listas en cuanto a la complejidad temporal, pues el análisis iba a ser el mismo.

Finalmente, analizando la forma en que se construye un algoritmo de Programación Dinámica (lo que veremos en el punto 4.2) llegamos a una solución del problema que nos permitió plantear la ecuación de recurrencia e implementarla en nuestro programa de tal forma que pudiésemos comparar los valores obtenidos con los esperados en las pruebas otorgadas por la cátedra, y así corroborar su funcionamiento, optimalidad y complejidad.

4.2. ¿Cómo encontrar la solución usando Programación Dinámica?

Al momento de plantear una solución a un problema utilizando programación dinámica debemos de poder identificar:

1. El/los caso/s base
2. La forma que tienen los subproblemas
3. La forma en que dichos subproblemas se componen para solucionar subproblemas más grandes

Luego de nuestro análisis logramos identificar claramente estos 3 puntos de nuestro problema:

1. Caso Base:

   Se presenta cuando no hay enemigos, es decir $n = 0$. En este caso, la mejor opción es no hacer nada, pues no tenemos enemigos a eliminar ni recargas que realizar.

2. La forma que tienen los subproblemas:

   Cargar o atacar en un minuto dado según la cantidad máxima de enemigos que podemos eliminar, teniendo en cuenta todas las posibles recargas que podemos considerar al atacar. O sea, buscamos el máximo de enemigos a eliminar comparando entre cargar, lo que es quedarnos con el máximo de enemigos eliminados anterior, y atacar, que es hallar el máximo de enemigos a eliminar, comparando cada posible suma de los enemigos eliminados para cierto j con el óptimo de enemigos eliminados si se carga de i a $i - j$.

3. La forma en que dichos subproblemas se componen para solucionar subproblemas más grandes:

   Al componer todos los subproblemas y tener un óptimo por cada minuto i hasta n, lo que sucede es que para el minuto n vamos a tener el óptimo que representa la solución del problema, y además, si nos guardamos el valor de j que se utilizó para cada óptimo, vamos a ser capaces de reconstruir la solución. Por ejemplo, si en el óptimo en n se atacó con un valor de recarga de $f(j)$, entonces de n a $n-j$ sabemos que se cargó, por lo que el próximo ataque estará en la posición $n - j$. Y así tendríamos que seguir el análisis hasta llegar al minuto 0, donde finalmente obtendremos la sucesión de ataques.

   De esta forma, para cada minuto i en la lista de llegadas de enemigos, se resuelve un subproblema que determina la cantidad máxima de enemigos que se pueden eliminar hasta ese minuto.

Ecuación de recurrencia

$\text{OPT}[i] = \text{máx}(\text{OPT}[i-1], \text{máx}_{1 \leq j \leq i}(\text{OPT}[i-j] + \text{mín}(\text{enemigos}[i-1], f(j-1))))$

Esta ecuación representa el óptimo en el minuto i, el cual es determinado tomando en cuenta el máximo entre dos opciones:

1. El óptimo previo, valuado en el tiempo $i - 1$.
2. El máximo valor obtenido, teniendo en cuenta todos los posibles tiempos de recarga anteriores, este tiempo esta simbolizado con la letra j y puede variar entre 1 e i.

4.3. Implementación del algoritmo

Para este informe trabajamos con el lenguaje Python. En nuestro archivo `main.py` se encuentra:

- La función para obtener los enemigos y las recargas de energías de los archivos
- La función que representa el algoritmo: `ataques_da_li(enemigos, recarga)`

A continuación presentamos el algoritmo de Programación Dinámica propuesto:

```
def ataques_da_li(enemigos, recarga):
    n = len(enemigos)
    OPT = [0] * (n + 1)
    recarga_usada = [0] * (n + 1)

    for i in range(1, n + 1):
        OPT[i] = OPT[i - 1]
        for r in range(1, i + 1):
            opt_actual = OPT[i]
            OPT[i] = max(opt_actual, OPT[i - r] + min(enemigos[i - 1], recarga[r -
    1]))
            if OPT[i] != opt_actual:
                recarga_usada[i] = r

    # Reconstruccion de la solucion
    secuencia_ataques = [CARGAR] * n
    i = n
    while i > 0:
```

```
        if recarga_usada[i] == 0:
            i -= 1
        else:
            secuencia_ataques[i - 1] = ATACAR
            i -= recarga_usada[i]

    return OPT[n], secuencia_ataques
```

Este consiste en:

- La lista `OPT` de tamaño $(n+1)$ se utilizará para almacenar las soluciones óptimas parciales del problema por cada minuto i, mientras que la lista `recarga_usada` (del mismo tamaño que `OPT`) se encargará de almacenar el índice de recarga utilizado en cada minuto, para luego así ser utilizada para construir la secuencia de ataques óptima.
- Para la búsqueda del óptimo se encuentran dos bucles: el primero itera sobre cada minuto que llamamos i, desde el minuto 1 hasta el minuto n (inclusive); el segundo bucle itera sobre los posibles tiempos de recarga que llamamos r, es decir, itera sobre las posibles recargas o ataques que se pudieron realizar previo al ataque analizado en el minuto correspondiente, lo cual implica recorrer desde el índice de recarga 1 hasta el minuto i (inclusive). Para cada i y r se calcula el valor óptimo en el minuto i considerando la opción de recarga en el minuto $i-r$ y luego atacar en el minuto i. Se elige el máximo entre el valor anterior en momento i y el valor obtenido con esta nueva opción.

```
    for i in range(1, n + 1):
        OPT[i] = OPT[i - 1]
        for r in range(1, i + 1):
            opt_actual = OPT[i]
            OPT[i] = max(opt_actual, OPT[i - r] + min(enemigos[i - 1], recarga
    [r - 1]))
            if OPT[i] != opt_actual:
                recarga_usada[i] = r
```

- Para la reconstrucción de la solución se inicializa una lista `secuencia_ataques` con todos los ataques en *Cargar*. Luego se empieza a recorrer la lista `recarga_usada` de manera inversa para reconstruir la secuencia de ataques. Si para un minuto i dado el arreglo tiene un valor de 0, significa que para ese i cargamos y tenemos que seguir avanzando. Si, en cambio, es distinto de 0, entonces significa que para ese minuto atacamos con un valor de recarga que corresponde al índice `recarga_usada[i] = r`, y debemos avanzar con i (de manera inversa) tantos minutos como r marque, pues para los minutos del medio sabemos que se va a estar cargando.

```
    secuencia_ataques = [CARGAR] * n
    i = n
    while i > 0:
        if recarga_usada[i] == 0:
            i -= 1
        else:
            secuencia_ataques[i - 1] = ATACAR
            i -= recarga_usada[i]
```

- Finalmente, devolvemos el máximo de enemigos eliminados (que se encuentra en la última posición de la lista) y la secuencia de ataques óptima para obtener ese máximo.

4.4. Complejidad

Al analizar la función que obtiene la máxima cantidad de enemigos eliminados, observamos primero que se crean dos listas de tamaño $(n+1)$, esto tiene una complejidad de tiempo y espacio de $\mathcal{O}(n)$. Luego se utilizan dos bucles, uno dentro del otro, que iteran ambos n veces en el peor de los casos, lo que resulta en una complejidad de $\mathcal{O}(n^2)$. Finalmente, para la reconstrucción de la secuencia de ataques, se recorre la lista de `secuencia_ataques` en el peor de los casos n veces, por lo tanto tiene una complejidad de $\mathcal{O}(n)$.

Haciendo los cálculos, queda:

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

Por lo que podemos concluir que la complejidad temporal total del algoritmo queda $\mathcal{O}(n^2)$.

4.5. Optimalidad

Dentro de las pruebas dadas por la cátedra pudimos chequear que efectivamente llegábamos a una solución óptima, existiendo casos donde la estrategia de ataques no era idéntica, pero el resultado en cuanto a enemigos totales eliminados era el correcto.

Pero en un caso general, podemos decir que el algoritmo propuesto es óptimo porque tiene una estrategia de exploración de todas las combinaciones posibles de ataques y recargas, maximizando la cantidad de enemigos eliminados para cada minuto i, por lo que no solo el algoritmo resulta óptimo para el problema general de n minutos, sino también para cada subproblema. Esto se cumple para cualquier valor de llegada de enemigos y de recargas (siempre y cuando cumplan las especificaciones de la consigna), por lo que la variabilidad de estos valores no afecta la optimalidad del algoritmo.

5. Mediciones

Llevamos a cabo mediciones del tiempo de ejecución del algoritmo para distintas cantidades de minutos n con un máximo de 5000, incrementando n de 10 en 10. Para las mediciones asignamos valores aleatorios tanto para las llegadas de enemigos por minuto como para las recargas, siempre y cuando los valores de esta última sean crecientes en el tiempo.

Registramos los resultados en un gráfico utilizando la biblioteca `matplotlib` de Python y los comparamos con una función de tendencia cuadrática para corroborar que se cumpla la complejidad teórica del algoritmo planteado, que dijimos es $\mathcal{O}(n^2)$.

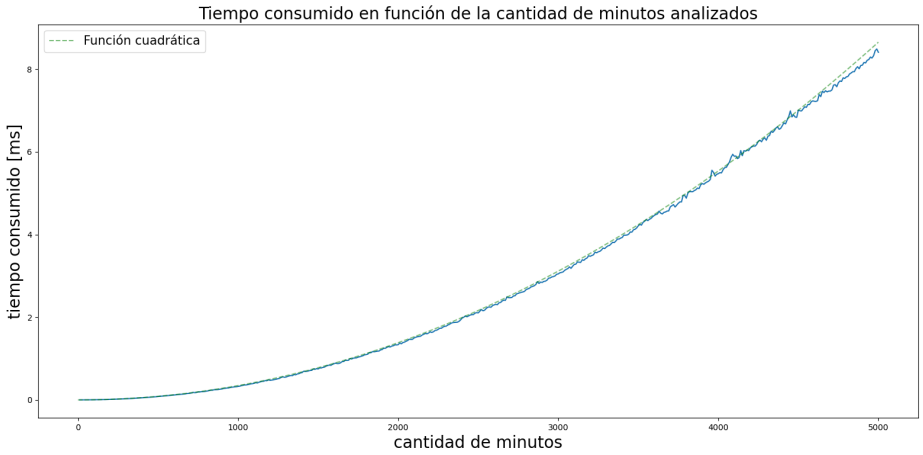

Gráfico 1: tiempo de ejecución del algoritmo en ms respecto a la cantidad de minutos comparado con una función de tendencia cuadrática.

Ambos gráficos presentan una similitud, aunque es esperable que no sean iguales, ya que los datos de tiempos de ejecución del algoritmo son experimentales y aproximados.

Por otro lado, también realizamos mediciones para un set de datos con solo los valores de las llegadas de enemigos por minuto variables, y para otro con solo las recargas por minuto variables.

Esto para analizar si (y cómo) afecta a los tiempos del algoritmo planteado la variabilidad de cada uno de esos valores.

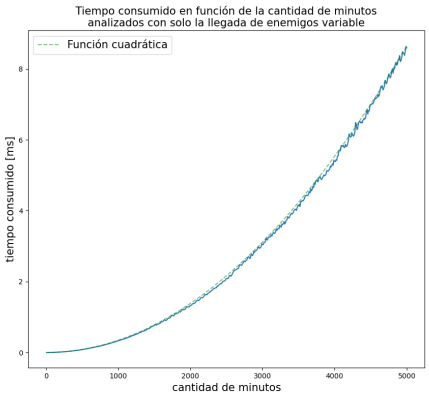

Gráfico 2: tiempo de ejecución del algoritmo en ms respecto a la cantidad de minutos.
Llegadas de los enemigos variable.

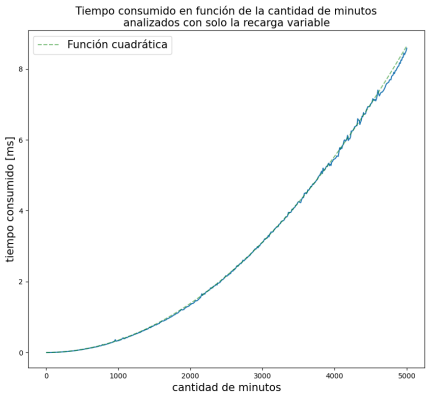

Gráfico 3: tiempo de ejecución del algoritmo en ms respecto a la cantidad de minutos.
Recargas variable.

Como podemos ver, los tiempos de ejecución del algoritmo no variaron significativamente entre ambos gráficos para ninguna cantidad de minutos en específico, es decir, para toda cantidad de minutos, el tiempo de ejecución del algoritmo fue parecido en ambos casos. Con esto podemos concluir que la variabilidad de los valores de las llegadas de enemigos y recargas no afecta a los tiempos del algoritmo planteado.

6. Conclusiones

La implementación del algoritmo propuesto nos permitió tener un mayor entendimiento del cómo funcionan los algoritmos de Programación Dinámica, desde su análisis hasta la demostración experimental de su complejidad mediante gráficos, comparados con una función de tendencia cuadrática correspondiente a la complejidad teórica calculada. Por otro lado, también pudimos establecer la optimalidad del algoritmo, que encuentra siempre la solución óptima al maximizar la cantidad de enemigos eliminados. Esto ya sea por comprobación utilizando las pruebas otorgadas por la cátedra, o también por el análisis mismo de la lógica de funcionamiento del algoritmo.