PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

IIC2283 Diseño y análisis de algoritmos
2° semestre 2019

Ayudantía 3

Dudas a Antonio López (alopez7@uc.cl)

Sección 1: Programación dinámica

1. Dispones de una secuencia de matrices $M_1, M_2, M_3, \ldots, M_n$ cuyas dimensiones son respectivamente $d_0 \times d_1, d_1 \times d_2, d_2 \times d_3, \ldots, d_{n-1} \times d_n$. Tu objetivo es multiplicar las matrices para obtener la matriz resultado de dimensiones $d_0 \times d_n$. El problema es que existen muchos órdenes en los que puedes multiplicar la secuencia llegando al mismo resultado pero la cantidad de multiplicaciones número a número varía. Crea un algoritmo que funcione con programación dinámica que determine el número mínimo de multiplicaciones necesarias para multiplicar toda la secuencia.
Considera que el número de multiplicaciones hechas al multiplicar dos matrices de $d_i \times d_{i+1}$ y $d_{i+1} \times d_{i+2}$ es $d_i \cdot d_{i+1} \cdot d_{i+2}$ y resulta en una matriz de $d_i \times d_{i+2}$.

   Por ejemplo $A \cdot B \cdot C$ con dimensiones $2 \times 4, 4 \times 6, 6 \times 2$ puede resolverse como $A \cdot (B \cdot C)$ realizando $4 \cdot 6 \cdot 2 + 2 \cdot 4 \cdot 2 = 64$ multiplicaciones o como $(A \cdot B) \cdot C$ realizando $2 \cdot 4 \cdot 6 + 2 \cdot 6 \cdot 2 = 72$ multiplicaciones.

   Solución: Dada una secuencia de matrices debemos agrupar las matrices de la forma en la que se hagan menos multiplicaciones, esto se ve en la siguiente recurrencia:

$$m(S,i,j) = \begin{cases} 0 & i = j \\ d_{i-1} \cdot d_i \cdot d_{i+1} & i+1 = j \\ \text{mín}_{k=i..j-1}(m(S,i,k) + m(S,k+1,j) + d_{i-1} \cdot d_k \cdot d_j) & \text{else} \end{cases}$$

   Esta recurrencia agrupa las matrices de la manera óptima pero hace llamadas recursivas repetidas por lo que se puede hacer más eficiente usando Programación Dinámica:
Si agregamos una tabla M de $n \times n$ en la que almacenamos los resultados de las llamadas recursivas podemos calcular lo mismo en menos tiempo. Iniciamos la tabla con -1 en cada casilla y alteramos la recurrencia.

$$m(S,i,j,M) = \begin{cases} M[i][j] & M[i][j] \neq -1 \\ 0 & i = j \\ d_{i-1} \cdot d_i \cdot d_{i+1} & i+1 = j \\ \text{mín}_{k=i..j-1}(m(S,i,k) + m(S,k+1,j) + d_{i-1} \cdot d_k \cdot d_j) & \text{else} \end{cases}$$

   Antes de retornar almacenamos el valor calculado en la tabla M. Además es posible crear una tabla secundaria O de $n \times n$ que almacene el k seleccionado en las llamadas recursivas de la recurrencia. Esta

tabla permite saber la forma en la que se agrupan las matrices para obtener el mínimo de multiplicaciones almacenado en M. El código del programa se encuentra en SIDING.

Finalmente se puede implementar esto de manera iterativa: las tablas M y O se llenan desde la diagonal central avanzando de diagonal en diagonal hasta la esquina superior derecha donde $i = 0$ y $j = n - 1$. El código de la versión iterativa también se encuentra en SIDING.

2. El problema del vendedor ambulante consiste en encontrar el ciclo en un grafo que pase por todos los nodos solo una vez de menor costo. Este problema puede ser planteado con programación dinámica.

   a) Escribe el algoritmo descrito usando programación dinámica.

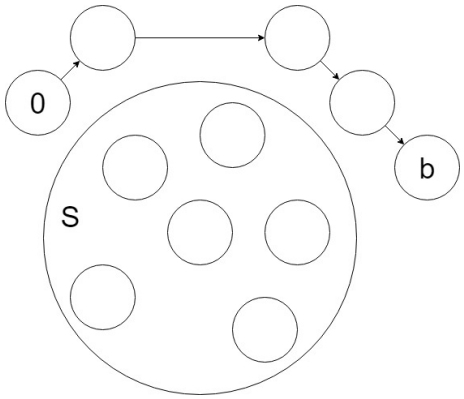
   Solución: Para usar programación dinámica en el problema hay que tratar de construir el ciclo de costo mínimo a partir de partes más pequeñas del problema. Un ciclo que pasa por todos los nodos se puede pensar como un camino que parte desde un nodo específico y pasa por todos los nodos para luego volver al nodo inicial. No importa el nodo inicial seleccionado ya que esto se cumple para todos los nodos por lo tanto diremos que el ciclo parte desde el nodo 0.

   Dado que el ciclo parte desde el nodo 0 debemos construir el resto del ciclo agregando el resto de los nodos al menor costo posible. Para eso consideraremos que el grafo está representado con una matriz de adyacencia M en la cual $M[i][j]$ indica el costo de la arista que conecta el nodo i con el nodo j. Si no existe una arista entre i y j entonces $M[i][j] = \infty$.

   Entonces si seleccionamos un nodo v cualquiera distinto del nodo 0 para continuar el camino tendríamos armado un tramo de ciclo y solo nos faltaría agregar los $n - 2$ nodos restantes. Esto es un subproblema más pequeño que el original que nos ayuda a construir el ciclo final. El planteamiento recursivo del vendedor ambulante es entonces:

$$\text{VA}(S, b) = \begin{cases} M[b][0] & S = \emptyset \\ \text{mín}_{v \in S}(\text{VA}(S - \{v\}, v) + M[b][v]) & \text{else} \end{cases}$$

   S es un subconjunto de los nodos del grafo y corresponde a todos los nodos que aún no se han incluido en el ciclo. b es el último nodo agregado al camino. $\text{VA}(S, b)$ es la función que retorna el mínimo costo con que se puede cerrar el ciclo que comienza con el nodo 0 y que debe ser completado con los nodos en S:



   Esta recurrencia se puede implementar en código obteniendo así un algoritmo que encuentra el mínimo costo del ciclo del vendedor ambulante. Para esto se representa el conjunto S con una

secuencia de bits tal que el nodo i está en el conjunto S solo si el i-ésimo bit es un 1. Este algoritmo tendrá muchas llamadas recursivas repetidas lo que hará innecesariamente ineficiente al algoritmo.

Para hacerlo más eficiente se usa la técnica de programación dinámica almacenando el resultado de cada llamada recursiva del algoritmo. Para esto se crea una tabla T de $2^n \times n$ en la que se almacenan los resultados ya calculados. En $T[S][b]$ se almacena el resultado de la llamada recursiva $\text{VA}(S, b)$

b) ¿Qué complejidad tiene el algoritmo que tienes en mente para este problema?

Solución: En el peor caso el algoritmo llenará la tabla T para llegar a la solución. Para esto debe hacer $2^n \cdot n$ llamadas a la función VA. Además cada llamada a la función VA itera sobre todos los nodos en S por lo que la complejidad final del algoritmo pertenece a $O(2^n \cdot n^2)$

c) ¿Cuanta memoria usa la solución propuesta?

Solución: La memoria usada corresponde a la tabla T y pertenece a $O(2^n \cdot n)$

Sección 2: Algoritmos codiciosos

1. Dada una secuencia $Q = q_1, \ldots, q_\ell$ de números racionales y una secuencia $I = [a_1, b_1], \ldots, [a_m, b_m]$ de intervalos de números racionales, decimos que $M \subseteq I$ es un cubrimiento para Q si:

$$Q \subseteq \bigcup_{j \in M} j$$

Además decimos que M es un cubrimiento mínimo para Q si para todo cubrimiento M' de Q se cumple que $|M| \leq |M'|$

a) Construye un algoritmo codicioso que funcione en tiempo polinomial y que recibe una secuencia Q y una secuencia I y retorna un cubrimiento mínimo para Q.

Solución:

- Ordenar la secuencia Q
- $M = \emptyset$
- Mientras $Q \neq \emptyset$:
  - $L = \{j \in I | Q[0] \in j\}$
  - Si $L = \emptyset$ return -1 (no existe cubrimiento)
  - Seleccionar elemento $l \in L$ que cubre más elementos de Q
  - $M = M \cup \{l\}$
  - $Q = Q - \{q \in Q | q \in l\}$
  - $I = I - \{l\}$
- return M

b) Demuestra que el algoritmo funciona correctamente.

Solución: Asumiremos que existe un cubrimiento para Q y demostraremos que el elemento agregado en la primera iteración pertenece a un cubrimiento mínimo. Luego demostraremos que el elemento seleccionado por la iteración i del programa pertenece al cubrimiento mínimo dados los elementos seleccionados en las iteraciones anteriores.

En la primera iteración se selecciona el subconjunto L de I. Este subconjunto incluye todos los intervalos que cubren el primer elemento de Q (el menor de todos). Si existe un cubrimiento para Q entonces obligatoriamente debe haber algún elemento de L ya que el primer elemento de Q debe ser cubierto. De todos los elementos de L se selecciona l el que cubre más elementos de Q. Ya que todos los elementos de L cubren a $Q[0]$ entonces el elemento l cubre a todo q cubierto por algún otro elemento de L. Por lo tanto si existe un cubrimiento mínimo que incluye un elemento $l' \in L$ tal que $l' \neq l$ entonces se puede remplazar por l generando otro cubrimiento mínimo. Por lo tanto l pertenece a algún cubrimiento mínimo.

Demostraremos que el elemento incluido a M en la iteración i del algoritmo es parte de un cubrimiento mínimo que incluye todos los elementos agregados a M en la iteraciones anteriores. Para esto asumiremos que los elementos agregados en las iteraciones 1 hasta la $i-1$ son un subconjunto de un cubrimiento mínimo para Q.

Dado el funcionamiento del algoritmo, el conjunto Q ahora contiene todos los elementos que no han sido cubiertos por ningún intervalo ya agregado a M, por o tanto es necesario incluir algún intervalo que cubra al primer elemento de Q. Se selecciona el conjunto L de intervalos que cubren a $Q[0]$, alguno de ellos tiene que pertenecer al cubrimiento mínimo final que incluye a los elementos ya agregados a M ya que o sino no se estaría cubriendo a todos los valores.

De este conjunto L se selecciona el elemento l que cubre más elementos de Q, por lo tanto si existe otro elemento $l' \in L, l' \neq l$ tal que pertenece a un cubrimiento mínimo que incluye a los elementos ya agregados a M, entonces existe otro cubrimiento mínimo en el cual remplazamos l' por l. Por lo tanto cuando incluimos l a M, M sigue siendo subconjunto de algún cubrimiento mínimo.

Finalmente cuando $Q = \emptyset$, todos los elementos están cubiertos por lo que M es un cubrimiento y además es mínimo.

c) Calcula la complejidad del algoritmo.

Solución: Las operaciones anteriores al loop toman tiempo polinomial. Todas las operaciones realizadas en el loop pueden ser realizadas en tiempo polinomial, y el loop se se repite a lo más $|Q|$ veces. Por lo tanto el algoritmo toma tiempo polinomial.