Programación dinámica II

Agenda
- i) PD's simples como TADs (conceptual)
- ii) Reconstrucción de soluciones
- iii) PD bottom-up
- iv) Pros y cons de bottom-up
- v) Factibilidad de PD
- vi) Ejercicio integrador

PD's simples como TADs

Problema del CD: dada una lista $d_1, \dots, d_n$ de duraciones de canciones y una capacidad k de un CD, encontrar el subconjunto D de canciones con máxima $\sum_{d \in D} d \leq k$

$cd : \{0, \dots, n\} \times \{-\infty, k\} \to \{-\infty, k\}$ tal que $cd(i,k)$ denota el óptimo para $d_1, \dots, d_i$ y k.

$$cd(i,k) = \begin{cases} -\infty & k < 0 \\ 0 & k \geq 0,\ i = 0 \\ \max(cd(i-1,k),\ d_i + cd(i-1, k-d_i)) & cc \end{cases}$$

Diccionario de memoización: matriz CD de $n \times (k+1)$

$CD[i,k] = cd(i,k)$ para todo $1 \leq i \leq n$, $0 \leq k \leq K$.

cuando $CD[i,k] \neq \perp$

Observación: la estructura CD nos permite resolver el problema para todo prefijo $d_1, \dots, d_i$ de canciones y toda duración $k \leq K$

TAD CD (conceptual)

- construir (D, k): crea el optimizador
- optimo (i, k): retorna $cd(i,k)$
- solucion(i,k): retorna una solución de valor $cd(i,k)$.

Estructura CD: con invariante $CD[i,k] \in \{\perp, cd(i,k)\}$
D: secuencia de canciones

Implementación top-down

construir (D, k):
    this->CD = [$\perp$ $\forall\, 1 \le i \le |D|,\ 0 \le k \le K$
    this->D = D
complejidad: $O(nk)$

optimo(i, k):
    si $k < 0$: retornar $-\infty$
    si $i = 0$: retornar 0
    si $CD[i,k] = \perp$:
        $CD[i,k] = \max\begin{cases} \text{optimo}(i-1, k), \\ D[i] + \text{optimo}(i-1, k - D[i]) \end{cases}$
    retornar $CD[i, k]$
complejidad: $O(1)$ amortizado (¿por qué?)

Reconstrucción de la solución

Idea: repasar la corrección de $cd(i,k)$.

i) $cd(i, k<0) = -\infty$ porque el problema está indefinido cuando la capacidad del CD es negativa.
$\Rightarrow$ Precondición: $k \geq 0$

ii) $cd(0, k \geq 0) = 0$ porque no puedo cargar canciones si no hay
$\Rightarrow$ si $i = 0$, retornar $\emptyset$

iii) $cd(i>0, k \geq 0) = \max\{cd(i-1, k), d_i + cd(i-1, k-d_i)\}$
porque hay dos alternativas para la canción i en una solución óptima S
a) $i \notin S \Rightarrow S$ es solución optima para $i-1, k$
b) $i \in S \Rightarrow S = S' \cup \{i\}$ y S' es óptima para $i-1, k-d_i$

$\Rightarrow$ solucion(i, k): // prec: $k \geq 0$
    si $i = 0$: retornar $\emptyset$
    si optimo$(i-1, k)$ = optimo(i, k):
        retornar solucion$(i-1, k)$
    retornar solucion$(i-1, k-D[i]) \cup \{i\}$

complejidad: $O(k)$ amortizado (se puede mantener para devolver en $O(1)$ amortizado)

Obs: solucion se puede computar fuera del TAD porque no necesita acceso a D

Ejemplo 2: Rod-cutting. Varilla de longitud n y precio p_i de vender varilla de longitud $1 \leq i \leq n$. Maximizar precio de venta total cortando la varilla

$rc: \{0, n\} \to \mathbb{N}$ tq $rc(k)$ es el óptimo cuando la varilla mide k

$rc(k) = \max \{0\} \cup \{ p_i + rc(k-i) \mid 1 \leq i \leq k \}$

Diccionario de memoización: vector RC de tamaño $n+1$ tq $RC[i] \in \{\bot, rc(i)\}$ $\forall$ $0 \leq i \leq n$

inicializar(P): $RC = \{ \bot \mid 0 \leq i \leq n \}$

complejidad: $O(n^2)$ amortizado (¿por qué?)

$rc(k)$: si $RC[k] = \bot$:
$RC[k] = \max \{0\} \cup \{ rc(k-i) \mid 1 \leq i \leq k \}$
retornar $RC[k]$

complejidad: $O(1)$ amortizado. (¿por qué?)

cortes(k)
$c = \arg\max(\{0 \to 0\} \cup i \to rc(k-i) \mid 1 \leq i \leq k)$
si $c > 0$: retornar $c \cup$ cortes$(k-c)$
sino: retornar $\emptyset$

complejidad: $O(k^2)$. Se puede llevar a $O(1)$ amortizado (¿cómo? ¿vale la pena?)

PD bottom-up

$$cd(i,k) = \begin{cases} -\infty & k<0 \\ 0 & k\geq 0,\ i=0 \\ \max(cd(i-1,k),\ d_i + cd(i-1,k-d_i)) & cc \end{cases}$$

⇒ Relación de precedencia entre instancias I y J:
$I \prec J$ I aparece en un llamado recursivo al cálculo J (orden parcial)

⇒ Orden topológico de instancias: es un orden lineal
$I_1 < \ldots < I_i < \ldots$ de las instancias tq
$I_i < I_j$ para todo $I_i \prec I_j$

Ejemplos: $(i,k) <_1 (j,h)$ lexicográfico
$(i,k) <_2 (j,h)$ si $i+k < j+h$
$(i,k) <_3 (j,h)$ si $(k,i) <_{lex} (h,j)$
etc

Recursión + orden topológico ⇒ algoritmo iterativo bottom up

Computar la función en el orden topológico hasta llegar al valor deseado, guardando los resultados intermedios. En lugar de invocar la función en forma recursiva, simplemente entrar a la estructura que guarda los resultados intermedios

PD: los resultados intermedios están en el diccionario de memoización.

CD bottom-up con orden lexicográfico en (i, k).

- Si $i=0 \Rightarrow CD[0,k] \stackrel{CD}{=} cd(0,k) \stackrel{cd}{=} 0$

- Si $i>0 \Rightarrow CD[i,k] \stackrel{CD}{=} cd(i,k) \stackrel{cd}{=}$
$\max(cd(i-1,k), d_i + cd(i-1, k-d_i)) \stackrel{CD+\alpha}{=}$
$\max(CD[i-1,k], d_i + CD[i-1, k-d_i])$

donde $CD[\cdot, k] = -\infty$ cuando $k < 0$.

Algoritmo

$cd(D, K)$

CD: matriz de $n+1 \times K$ con $CD[0,\cdot] = 0$ // base
para $i = 1,\dots,n$ para $k = 0,\dots,K$: // orden topo
$CD[i,k] = \max(CD[i-1,k], D[i] + CD[i-1, k-d_i])$
retornar $CD[n,K]$

<u>Obs:</u> Para el TAD CD la función anterior se usa para inicializar. La reconstrucción de la solución <u>no</u> cambia

Rod cutting

$$\overset{RC[u]}{rc(u)} = \max \{0\} \cup \{p_i + \overset{RC[u-i]}{rc(u-i)} \mid 1 \leq i \leq u\}$$

$\Rightarrow$ orden topológico: $0, \rightarrow m$.

$rc(P)$: $RC = \{0 \mid 0 \leq u \leq m\}$
para $i = 1, \rightarrow m$
$RC[i] = \max\{P[i] + RC[u-i] \mid 1 \leq i \leq u\}$
retornar $RC[m]$

Pros y contras de bottom-up.

+ Control sobre el orden de ejecución
− Necesidad de definir/computar el orden topológico
+ Posibilidad de reducir consumo de memoria
+ Optimizaciones de bajo nivel (cache, etc)
− El algoritmo resultante es menos declarativo
otras?

Factibilidad de PD

Selección de representantes: dadas n triplas con números entre 1 y $u \leq 3n$, queremos encontrar un subconjunto S tal que cada tripla tenga exactamente un elemento de S

Ejemplo: $\{1,2,3\}$ $\{2,3,4\}$ $\{1,3,5\}$ ✓
$\{1,2,3\}$ $\{2,3,4\}$ $\{1,3,4\}$ $\{1,2,4\}$ ✗

Idea para recursión: considero cada elemento y veo si está o no en la solución (misma idea que CD)

Sea $\mathcal{S}$ la familia de subconjuntos de $\{1, \dots, u\}$
$\Rightarrow f : \{1, \dots, u\} \times \mathcal{S} \to \{T, F\}$ tq $f(i, S) = T$ si existe una solución incluida en $\{1, \dots, i\} \cup S$ que contenga a S

$$f(i,S) = \begin{cases} F & \text{si } (i=0 \wedge S \cap t = \emptyset) \vee |S \cap t| > 1 \text{ para } t\text{: tripla} \\ T & \text{si } |S \cap t| = 1 \text{ para toda tripla} \\ f(i-1, S) \vee f(i-1, S \cup \{i\}) & \text{cc} \end{cases}$$

1) ¿satisface propiedad de superposición de subproblemas?
2) ¿Vale la pena memoizar? 3) ¿Alguna instancia se resuelve más de una vez?

Ejercicio integrador (UVA 10261, Ferry loading)

Cola $p_1, \ldots, p_n$ de pesos y dos mochilas M_1 y M_2 de capacidad K. Queremos meter la mayor cantidad de pesos en cada mochila respetando el orden de la cola

Ejemplo: cola $\langle 9, 3, 6, 1, 4, 1, 1, 3, 1 \rangle$ y capacidad 10
=> se agregan los pesos 9, 3, 6, 1

Solución 1

$f(i, K_1, K_2)$ = máxima cantidad de pesos en $\{i, \ldots, n\}$ cuando las capacidades son K_1 y K_2

$$f(i, K_1, K_2) = \begin{cases} -\infty & \text{si } K_1 < 0 \vee K_2 < 0 \\ 0 & \text{si } i = n+1 \text{ y } K_1, K_2 \geq 0 \\ \max\{0, 1 + f(i+1, K_1 - p_i, K_2), 1 + f(i+1, K_1, K_2 - p_i)\} & \text{cc} \end{cases}$$

=> complejidad $O(n K^2)$

Solución 2

Notar que no todas las instancias son válidas
Ej: si $Q = \langle 9, 3, \ldots \rangle$ y $K = 10$ => es imposible que ambas mochilas tengan capacidad 5.

$f(i, K_1)$ = máxima cantidad de pesos en $\{i, \ldots, n\}$ cuando la mochila 1 tiene capacidad K_1 y la mochila 2 tiene ocupada la capacidad necesaria para poner los primeros $i-1$ elementos. Es decir

$$K_2 = K - \left(\sum_{j=1}^{i-1} p_j - (K - K_1)\right) = 2K - \sum_{j=1}^{i-1} p_j - K_1$$

$$f(i, k_1, \cancel{k_2}) = \begin{cases} -\infty & \text{si } k_1 < 0 \vee k_2 < 0 \\ 0 & \text{si } i = n+1 \text{ y } k_1, k_2 \geq 0 \\ \max\{0, f(i+1, k_1 - p_i, \cancel{k_2}), f(i+1, k_1, \cancel{k_2 - p_i})\} \end{cases}$$

donde $k_2 = 2k - k_1 - P[i-1]$ y $P[i] = \sum_{j=1}^{i} p_j$ precalculado

complejidad: $O(n \cdot k)$

Solución 3 (avanzada)

Supongamos que se pueden agregar i pesos para algún i. De todas las soluciones, se puede tomar la que maximiza el peso total

$\Rightarrow$ mochila 1 tiene un peso igual al del problema de knapsack para capacidad k, costos $p_1, \ldots, p_i$ y beneficios $p_1, \ldots, p_i$.
mochila 2 tiene peso $k - (P[i] - M_1)$ donde M_1 es el peso de la mochila 1.

```
ferry-loading(n, k):
    i = 1
    P[i] = Σ_{j=1}^{i} p_j   ∀ 1 ≤ i ≤ n
    mientras i ≤ n ∧ k - P[i] + knapsack(i, k) ≥ 0
        i += 1
    retornar i
```

complejidad: $O(nk)$. Recordar que knapsack(i, k) cuesta $O(1)$ amortizado y knapsack(n, k) se inicializa en $O(nk)$ tiempo