Resumen Control 6

Programación Dinámica

Introducción

Es un método para reducir el tiempo de ejecución de un algoritmo mediante el uso de *subproblemas superpuestos* y *subestructuras óptimas*, es decir, el uso de soluciones óptimas de subproblemas para encontrar la solución óptima del problema en su conjunto.

Selección de tareas con ganancias

Problema:

- N tareas, en donde cada una tiene una hora de inicio s_i y hora de término t_i que definen el intervalo de tiempo $[s_i, t_i)$ de la tarea.
- Máquina única que realiza una tarea a la vez, por lo que las tareas deberán esperar si sus intervalos de tiempo se traslapan.
- Cada tarea produce una ganancia v_i si es realizada, por lo que no importa el número de tareas a realizar, sino cuáles tareas se deben realizar para que la suma de ganancias sea máxima.

Suponemos que las tareas están ordenadas por hora de término:

- $f_1 \leq f_2 \leq \ldots \leq f_n$

Para cada tarea j definimos $b(j)$ como la tarea i que termina más tarde antes del inicio de j

- $f_i \leq s_j$ tal que para todo $k > i$, $f_k > s_j$
- $b(j) = 0$ si ninguna tarea $i < j$ satisface la condición anterior

Sea Ω una solución óptima:

1. Si la tarea n no pertenece a Ω, entonces Ω es igual a la solución óptima para las tareas $1,\ldots,n-1$.
2. Si la tarea n pertenece a Ω, entonces ninguna tarea k, $b(n) < k < n$, puede pertenecer a Ω. Además, Ω debe incluir una solución óptima para las tareas $1,\ldots,b(n)$.

Es por lo anterior, que la solución óptima para las tareas $1,\ldots,n$ involucra encontrar soluciones óptimas a problemas más pequeños del mismo tipo. Ahora, si Ω_j es la solución óptima al problema de las tareas $1,\ldots,n$ y $opt(j)$ es su valor, entonces buscamos Ω_n con valor $opt(n)$.

Generalizando:

- j pertenece a Ω_j , en cuyo caso opt(j) = v_j + opt(b(j))
- j no pertenece a Ω_j , en cuyo caso opt(j) = opt(j–1)

Por lo que,

$$\text{opt}(j) = \max\{ v_j + \text{opt}(\text{b}(j)) , \text{opt}(j-1) \} \qquad (*)$$

Dado a que todas las tareas están ordenadas por hora de término y que se tienen calculados los `b(j)` para cada `j`, y se tiene `opt(0)` para el óptimo de un conjunto vacío de tareas, es posible calcular `opt(n)` mediante el siguiente algoritmo recursivo:

```
opt(j):
    if j = 0:
        return 0
    else:
        return max{ v_j + opt(b(j)) , opt(j-1) }
```

Sin embargo, su tiempo de ejecución en el peor caso origina otras dos llamadas a `opt` resolviendo `n + 1` subproblemas diferentes, pero esto toma <u>tiempo exponencial</u> ya que resuelve múltiples veces cada subproblema. Lo anterior se puede modificar mediante el almacenamiento de `opt(j)` en un arreglo global la primera vez que se calcula:

```
rec-opt(j):
    if j = 0:
        return 0
    else:
        if m[j] no está vacía:
            return m[j]
        else:
            m[j] = max{ v_j + rec-opt(j) , rec-opt(j-1) }
            return m[j]
```

La nueva complejidad es `O(n)`. Sin embargo, además de calcular el valor de la solución óptima, hay que saber cuál es la solución. Para esto, solo debemos llenar el arreglo `m`, en donde `m[n]` contendrá el valor de la solución óptima. También, es posible calcular los valores en m iterativamente en tiempo `O(n)`:

```
it-opt:
    m[0] = 0
    for j = 1, 2, ..., n :
        m[j] = max{ v_j + m[b(j)] , m[j-1] }
```

Propiedades de un algoritmo de programación dinámica

Para desarrollar un algoritmo de programación dinámica se requiere de una colección de subproblemas, derivados del problema original que satisfagan las siguientes propiedades:

- Número de subproblemas polinomial (idealmente)
- Solución del problema original debe poder calcularse a partir de las soluciones a los subproblemas
- Existe un orden natural de los subproblemas, desde "el más pequeño" hasta "el más grande"
- Recurrencia fácil que permita calcular la solución a un subproblema a partir de las soluciones de subproblemas más pequeños

La mochila con objetos 0/1

Tenemos n objetos no fraccionables, cada uno con un valor v_k y peso w_k. Queremos ponerlos en una mochila con capacidad total $W < \sum w_k$ (no podemos incluirlos a todos) de manera que se maximice la suma de los valores.

Si $knap(p, q, \omega)$ representa el problema de maximizar $\sum v_k x_k$

… sujeto a $\sum w_k x_k \leq \omega$ y $x_k = \{0, 1\}$

… entonces el problema es $knap(1, n, W)$

Sea $y_1, \ldots, y_n$ una selección óptima de valores 0/1 para $x_1, \ldots, x_n$. Podríamos tener que $y_1 = 0$ ó 1:

1. Si $y_1 = 0$, entonces el objeto 1 no está en la solución óptima
2. Si $y_2 = 1$, entonces el objeto 1 está en la solución óptima

Luego, se debe cumplir que $y_2, \ldots, y_n$ deba ser una selección óptima para $knap(2, n, W)$ ya que en caso contrario no habría selección óptima para $knap(1, n, W)$. En cambio, para el segundo caso, $y_2, \ldots, y_n$ debe ser selección óptima para $knap(2, n, W - w_1)$ porque si no sería una selección con mayor valor. Ergo, el problema se puede resolver a partir de las soluciones óptimas de subproblemas del mismo tipo.

Sea $g_k(w)$ el valor de la solución óptima a $knap(k + 1, n, w)$:

- $g_0(W)$ es el valor de la solución óptima a $knap(1, n, W)$ – el problema original
- $g_0(W) = \max\{g_1(W), g_1(W - w_1) + v_1\}$

... si $y_1, y_2, \ldots, y_n$ es una solución óptima a knap(1, n, W),

... entonces para cada j, $1 \le j \le n$

$$y_1, \ldots, y_j \;,\; y_{j+1}, \ldots, y_n$$

... deben ser soluciones óptimas a[1]

$$\text{knap}(1, j, \textstyle\sum w_k y_k)\;,\; 1 \le k \le j$$

$$\text{knap}(j+1, n, W - \textstyle\sum w_k y_k),\; 1 \le k \le j$$

Luego, se llega hasta `gₙ(w) = 0 if w = 0, gₙ(w) = -∞ if w < 0`. A continuación se muestra un ejemplo:

P.ej., si $n = 3$, $(w_1, w_2, w_3) = (2, 3, 4)$, $(v_1, v_2, v_3) = (1, 2, 5)$, y $W = 6$

... tenemos que calcular

$$g_0(6) = \max\{\, g_1(6), g_1(4)+1 \,\}$$

$g_1(6) = \max\{\, g_2(6), g_2(3)+2 \,\} = \max\{5, 2\} = 5$, ya que

$$g_2(6) = \max\{\, g_3(6), g_3(2)+5 \,\} = \max\{0, 5\} = 5$$

$$g_2(3) = \max\{\, g_3(3), g_3(-1)+5 \,\} = \max\{0, -\infty\} = 0$$

$g_1(4) = \max\{\, g_2(4), g_2(1)+2 \,\} = \max\{5, 2\} = 5$, ya que

$$g_2(4) = \max\{\, g_3(4), g_3(0)+5 \,\} = \max\{0, 5\} = 5$$

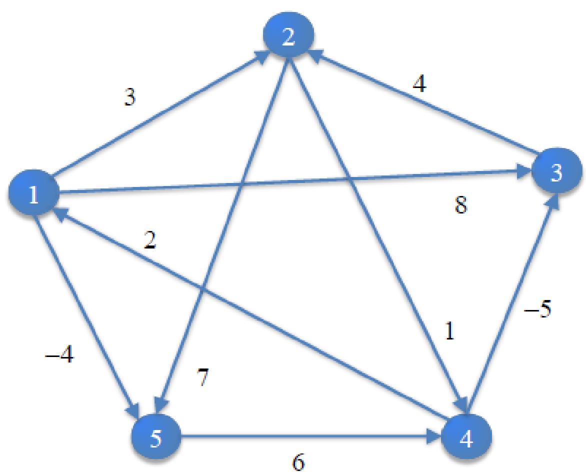
$$g_2(1) = \max\{\, g_3(1), g_3(-3)+5 \,\} = \max\{0, -\infty\} = 0$$

Luego, $g_0(6) = \max\{5, 5 + 1\} = 6$

Rutas más cortas entre todos los pares de vértices

Vimos Dijkstra para el caso en que los costos de las aristas son no negativos cuyo tiempo de ejecución es $O(VE\log V)$, en caso de que puedan tener costos negativos es posible usar el algoritmo de Bellman – Ford cuyo tiempo de ejecución seria normalmente $O(V^2E)$, mientras que en grafos densos es $O(V^4)$. Sin embargo, dichas complejidades se podrían mejorar mediante programación dinámica.

Emplearemos la representación de un grafo G mediante su matriz de adyacencias, en donde sus vértices serán numerados del 1 al n (o sea, $|V| = n$), y el *input* es una matriz W que representa los costos de las aristas:



$$W = \begin{matrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{matrix}$$

De lo anterior se rescata de w_{ij} lo siguiente:

$$\begin{aligned} \omega_{ij} &= 0 && \text{si } i = j \\ &= \text{costo de la arista direccional } (i, j) && \text{si } i \neq j \text{ y } (i, j) \in E \\ &= \infty && \text{si } i \neq j \text{ y } (i, j) \notin E \end{aligned}$$

Además, se supone que G no contiene ciclos de costo negativo.

Algoritmo Floyd – Warshall

Es un algoritmo para encontrar las rutas más cortas que considera los vértices intermedios de una ruta más corta. Si G es un grafo con vértices V = {1,…,n} y sea el subconjunto {1,…,k} para algún k, consideremos cualquier par de vértices (i, j) ∈ V, y todas las rutas de i a j cuyos vértices intermedios están todos tomados del conjunto {1,…,k}, en donde p es la ruta más corta entre ellas y k puede o no ser un vértice (intermedio) de p:

- Si k no es vértice de p, entonces todos los vértices (intermedios) de p están en el conjunto {1,…,k - 1}, por lo que la ruta más corta de i a j con todos los vértices intermedios en {1,…,k - 1} es la misma que para {1,…,k}
- Si k es vértice de p, entonces podemos dividir p en dos tramos: p_1 de i a k y p_2 de k a j, en donde ambos, por optimalidad, son las rutas más cortas

Si se define $d_{ij}^{(k)}$ el costo de una ruta más corta de i a j tal que todos los vértices intermedios están en el conjunto {1,…,k}. Cuando k = 0, una ruta de i a j sin vértices intermedios con número mayor que 0 tiene a lo más una arista $d_{ij}^{(0)} = w_{ij}$, esto es:

$$
\begin{aligned}
d_{ij}^{(k)} &= \omega_{ij} && \text{si } k = 0 \\
&= \min\left(d_{ij}^{(k-1)} , d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \right) && \text{si } k \geq 1
\end{aligned}
$$

Por lo tanto, la matriz $D^{(n)} = d_{ij}^{(n)}$ entrega la respuesta final:

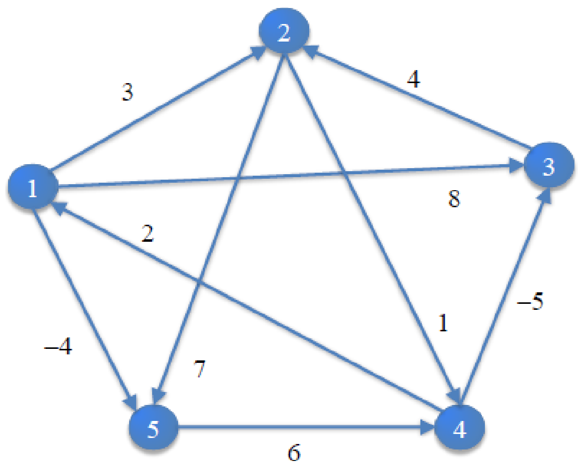
$$d_{ij}^{(n)} = \delta(i, j) \text{ para todo } i, j \in V$$

El algoritmo de Floyd-Warshall, *bottom-up*, toma tiempo $O(V^3)$

```
D(0) = W
for k = 1 … n:
    sea D(k) = (dij(k)) una nueva matriz
    for i = 1 … n:
        for j = 1 … n:
            dij(k) = min(dij(k-1), dik(k-1)+dkj(k-1))
return D(n)
```

Ejemplo:

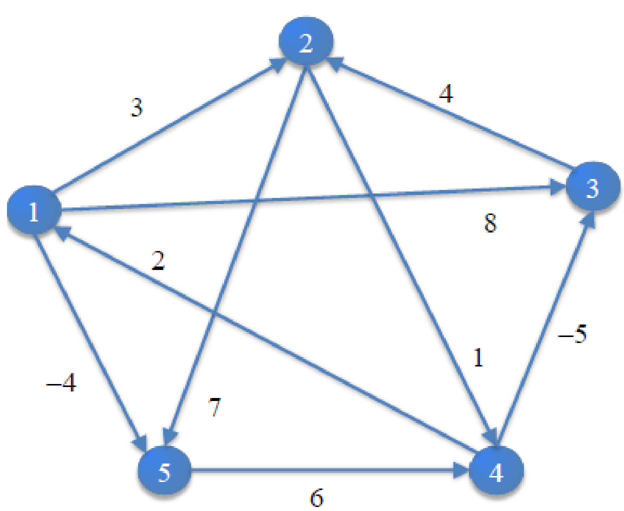
$$D^{(0)} = \begin{matrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{matrix}$$



$$D^{(1)} = \begin{matrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{matrix}$$

$$D^{(2)} = \begin{matrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{matrix}$$

$$D^{(3)} = \begin{matrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{matrix}$$



$$D^{(4)} = \begin{matrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{matrix}$$

$$D^{(5)} = \begin{matrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{matrix}$$