Algorísmia QP 2019–2020

Segon control online **04 de Mayo de 2020**

Durada: 1h

Instruccions generals:

- Heu de lliurar les respostes en un únic arxiu pdf a través del Racó. No oblideu posar el vostre nom. Indiqueu clarament la pregunta que esteu responent.
- Heu d'argumentar la correctesa i l'eficiència dels algorismes que proposeu. Per això podeu donar una descripció d'alt nivell de l'algorisme amb les explicacions i aclariments oportuns que permetin concloure que l'algorisme és correcte i té el cost indicat.
- Heu de justificar totes les vostres afirmacions, en cas contrari la nota de la pregunta serà 0. Quan la pregunta sigui una qüestió (Cert o Fals?), heu de dir si l'enunciat és cert o fals abans de justificar la resposta.
- Podeu fer crides a algorismes que s'han vist a classe, però si la solució és una variació n'haureu de donar els detalls.
- Es valorarà especialment la claredat i concisió de la presentació.
- La puntuació total d'aquest control és de **10 punts**.
- Assumim que us comprometeu a realitzar aquest control vosaltres mateixos sense consultar material ni els companys.

Exercici 1 (0.75 punts) Cierto o Falso?: El tiempo para resolver un problema de programación dinámica siempre es $\Theta(k)$, donde k es el número de subproblemas diferentes del problema que aparecen en la recurrencia.

Una solución Falso. En PD resolveremos los k subproblemas pero el tiempo por subproblema puede ser cualquiera y no siempre es constante.

Exercici 2 (0.75 punts) Cierto o Falso?: Si $G = (V, E)$ es un grafo no dirigido y conexo con pesos $w : E \to \mathbb{Z}$, se pueden definir distancias entre los pares de vértices de G.

Una solución Falso. Solo cuando el grafo no tiene ciclos con peso negativo.

Exercici 3 (0.75 punts) Cierto o Falso?: El diámetro de un grafo $G = (E, V)$ es la profundidad del árbol que obtenemos al hacer una búsqueda en anchura (BFS), a partir de un vértice cualquiera $u \in V$.

Recordad que *el diametro* es el máximo de las distancias entre pares de vértices de G, i.e. G tiene diámetro d sii $\forall u, v \in V, \delta(u, v) \leq d$.

Una solución Falso. La altura del BFS nos da la distancia más larga del vértice origen. En el caso de un árbol binario completo en el BFS desde la raíz, hay vertices (pares de hojas) a distancia doble de la altura. En general si el grafo fuera un árbol el diámetro será la distancia entre las dos hojas más distantes a la raíz del BFS.

Exercici 4 (0.75 punts) Cierto o Falso?: Dado un grafo dirigido con pesos $G = (V, E)$ sin ciclos con peso negativo, el camino más corto, para todos los pares de vértices, se puede obtener en tiempo $O(|V|^3)$.

Una solución Cierto. Utilizando el algoritmo de Floyd-Warshall.

Exercici 5 (1.5 punts) Tenemos un grafo no dirigido y conexo $G = (V, E)$ con pesos $w : E \to \mathbb{Z}$, y tal que $|V| = n$ y $|E| = n$. Proporcionad un algoritmo con coste **lineal** tal que, dado un vértice s calcule las distancias.de s al resto de vértices o determine que esto no es posible.

Una solución Si el grafo tiene una arista $e = (u, v)$ con peso negativo, el ciclo $u-v-u$ tiene peso negativo. Como el grafo es conexo la arista es accesible desde s y no podemos definir distancias.

Si los pesos son no negativos, al tener n aristas el grafo solo puede contener un ciclo. Con un BFS podemos obtener la secuencia de vértices en este ciclo. Tenemos dos casos:

- s está en el ciclo. Podemos calcular la distancia de s a los vértices del ciclo avanzando desde s en las dos direcciones, y cortando el ciclo en el momento en que las distancias al vértice por los un camino sea mayor que por el otro. con coste $O(n)$.

  Después, eliminamos las aristas del ciclo, esto nos da un bosque de expansión. Con un BFS desde cada vértice del ciclo podemos obtener las distancias de s al

resto de vértices. Como en cada BFS exploramos conjuntos de aristas diferentes el coste total es $O(n+m) = O(n)$

- s no está en el ciclo. En ese caso hay un único camino que conecte s con un vértice v del vértice v que está más cercano s.

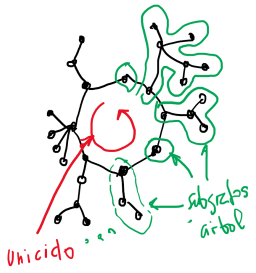
  Utilizariamos el mismo algoritmo que antes, pero ahora calcularíamos las distancias e v en el ciclo en vez de a s y las actualizariamos sumando la distancia de s a v.

  El resto de distancias se calcularian igual que en el paso anterior.

El coste total del algoritmo es $O(n)$.

Otra solución: Si existe una arista (u, v) de peso negativo en G finalizamos puesto que el ciclo $u - v - u$ es de peso negativo. Podemos determinar la existencia de una tal arista en tiempo $O(m) = O(n)$. En caso congtrario todos los pesos son positivos y las distancias están bien definidas.

Mediante un DFS o BFS con coste $O(n+m) = O(n)$ la única arista de retroceso en el grafo (dado que $|V| = |E| = n$ y que G es conexo, sabemos que G contiene exactamente un ciclo). Podemos considerar que el grafo está formado por un ciclo en el cual cada vértice de ciclo tiene adjuntado un subgrafo que es un árbol (ver figura).



De hecho el BFS/DFS simple con el que encontramos el ciclo único en G en el paso anterior puede 'etiquetar" cada vértice de tal modo que, con coste $O(n)$ tendremos, para cada $v \in G$, $en\_ciclo[v] =$ **true** ssi v es un vértice del ciclo.

Dados s y u entre s y u habrá exactamente uno o dos caminos en G; uno si s y u están en el mismo subárbol, y dos si s y u no están en el mismo subárbol.

Si s forma parte del ciclo entonces tiene dos vecinos en el ciclo $\ell = \ell(s)$ y $r = r(s)$ y los podemos hallar muy fácilmente gracias a en_ciclo. Si s no está en el ciclo, entonces está en un árbol y podemos buscar mediante un recorrido el vértice $p(s)$ que pertenece al ciclo y está en el mismo subárbol que s. Y hallar los vecinos $\ell = \ell(p(s))$ y $r = r(p(s))$. Si estamos en el primer caso (s está en el ciclo) retiramos la arista (s, ℓ) y calculamos el peso de los caminos desde s a los restantes vértices con un recorrido, digamos que se obtiene $\Delta_1[u] = \delta(s, u)$ para todo u en el grafo $G \setminus \{s, \ell\}$. De igual forma podemos obtener $\Delta_2[u]$ la distancia mínima de s a u para todo vértice u en el grafo G sin la

arista (s, r). La distancia mínima de s a u en G será, obviamente, $\min\{\Delta_1[u], \Delta_2[u]\}$ y la habremos podido obtener con dos recorridos con coste $O(n + m) = O(n)$. Si s no está en el ciclo, el algoritmo es el mismo, excepto que las aristas a eliminar son $(p(s), \ell)$ y $(p(s), r)$—si s y u están en el mismo subárbol entonces solo hay un camino y $\Delta_1[u] = \Delta_2[u]$, pero podemos aplicar el mismo algoritmo para calcular las distancias; el único análisis por casos que hay que hacer es para determinar que dos aristas del ciclo deben retirarse.

Exercici 6 (1.5 punts) Tenemos una red de comunicación formada por n servidores que representamos con un grafo dirigido acíclico $G = (V, E)$ con $|V| = n$ vértices y $|E| = m$ aristas. Para cada $e \in E$, $b(e) \geq 0$ es el *ancho de banda* de la arista e. El ancho de banda de un camino P es el minimo de los anchos de banda de las aristas que lo forman. Dados dos vértices s y t proporcionad un algoritmo con coste $O(n+m)$ para encontrar un camíno con anchura de banda máxima entre s y t.

Una solución

Como el grafo es dirigido y acíclico podemos utilizar el algoritmo específico de cálculo de distancias para DAGs que tiene coste $O(n + m)$. Tendremos que modificar el algoritmo para la función Relax, en vez de calcular la suma de los pesos calcule el mínimo. También calcular el max en vez del min en la PD asociada a cada nodo; es decir, haremos $D[v] = \max(D[v], \min(D[u], b(u, v)))$ en vez del habitual $D[v] = \min(D[v], D[u] + w(u, v))$ siendo $w(u, v)$ el peso de arco (u, v).

Exercici 7 (1.5 punts) Diseñad un algoritmo para resolver en tiempo polinómico el siguiente problema. Dado un grafo dirigido $G = (V, E)$ en el que los vértices son los enteros $\{1, 2, \ldots, n\}$ y dónde cada arista tiene un peso $w(e) \in \mathbb{Z}^+$ correspondiente al tráfico esperado en esa arista. Un vértice i controla el tráfico

$$T(i) = \sum_{(k,\ell)\in E, k<i<\ell} w(k, \ell) + \sum_{(\ell,k)\in E, k<i<\ell} w(\ell, k).$$

Proporcionar una recurrencia que os permita calcular $T(i)$ eficientemente.

Una solución

Notemos que $T(1) = 0$ y $T(n) = 0$. $T(i)$, $1 < i < n$, tiene dos partes,

$$T_1(i) = \sum_{(k,\ell)\in E, k<i<\ell} w(k, \ell),$$
$$T_2(i) = \sum_{(\ell,k)\in E, k<i<\ell} w(\ell, k).$$

$T_1(i)$ se puede obtener a partir de $T_1(i - 1)$ como

$$T_1(i) = T_1(i - 1) - \sum_{(k,i)\in E, k<i-1} w(k, i) + \sum_{(i-1,\ell)\in E, i<\ell} w(i - 1, \ell)$$

el primer bloque son las aristas que contaban con $i-1$ pero no con i y el segundo las que añadimos a la contabilidad de i. El resto de arista en $T_1(i-1)$ contabilizan tanto en $T_1(i-1)$ como en $T_1(i)$. Observemos que también $T_1(1) = 0$ y $T_1(n) = 0$.

La segunda parte es simétrica si tomamos los vértices en el orden inverso. Esto nos da

$$T_2(i) = T_2(i+1) - \sum_{(\ell,i)\in E, \ell > i+1} w(\ell, i) + \sum_{(i+1,k\}\in E, k<i} w(i+1, k)$$

Exercici 8 (2.5 punts) Dado un vector $A[1..n]$ de numeros enteros positivos, queremos hallar la subsecuencia ascendente mas larga en A. Una subsecuencia ascendente de A de longitud k es una secuencia $A[i_1] \leq A[i_2] \leq \cdots \leq A[i_k]$, donde $1 \leq i_1 < i_2 < \cdots < i_k \leq n$. Proporcionad un algoritmo de PD que resuelva el problema.

Una solución

Una solución óptima empieza en la posición i_1 y continúa con la subsecuencia más larga que se puede obtener empezando en la posición $i_2 > i_1$, donde se ha de cumplir que $A[i_1] \leq A[i_2]$.

A la vista de esta estructura de suboptimalidad consideramos el subproblema correspondiente: sea $C(i)$ la longitud de la subsecuencia creciente más larga que se puede obtener incluyendo $A[i]$.

Una vez calculados los $C(i)$'s la solución al problema será una secuencia con longitud

$$\max_{1\leq i\leq n} C(i).$$

El caso base es cuando $i = n$, ya que no podemos continuar la secuencia. Otro caso base se da cuando $A[i]$ es mayor que cualquier elemento a su derecha. Si convenimos que $A[n+1] = +\infty$ y que $C(n+1) = 0$ podremos escribir nuestra recurrencia—la que se deriva de la estructura de suboptimalidad comentada—de manera muy simple:

$$C(i) = 1 + \max\{C(k) \mid i < k \leq n+1 \wedge A[i] \leq A[k]\}, \qquad 1 \leq i \leq n.$$

Esta formulación recursiva se puede implementar mediante un recorrido de una tabla que guarda los $C(i)$'s desde la posición n a la 1 con coste total $O(n^2)$, pues necesitamos tiempo $O(n-i)$ para calcular $C(i)$.

Para poder construir la secuencia calcularemos también la matriz de punteros poniendo $D(n+1) = n+1$ y

$$D(i) = \arg\max\{C(k) \mid i < k \leq n+1 \wedge A[i] \leq A[k]\}, \qquad 1 \leq i \leq n.$$

Una vez localizado el valor i_1 que nos proporciona el máximo en $\max_{1\leq i\leq n} C(i)$, siguiendo los punteros hasta llegar a $n+1$ en D podemos reconstruir una solución óptima en tiempo adicional $O(n)$.

El coste total del algoritmo es $O(n^2)$ y utiliza espacio adicional $O(n)$.

Se puede mejorar el coste en tiempo de este algoritmo para que sea $O(n)$, tras un preproceso de coste $O(n \log n)$ y usando espacio auxiliar adicional lineal.