Diseño de algoritmos

Jesús Bermúdez de Andrés. UPV-EHU
Guías para la solución de ejercicios: Algoritmos voraces

Curso 2008-09

1. Considere la siguiente representación con listas de adyacencias del grafo no dirigido $G = (V, A)$. Un par $(w,\ p)$ en la lista de adyacencia del nodo v significa que existe una arista (v, w) con peso asociado p.

$V = [V(1), V(2), V(3), V(4), V(5), V(6), V(7)]$	
$V(1) = [(3,3), (4,10), (5,11)]$	$V(5) = [(1,11), (6,6)]$
$V(2) = [(3,8), (4,12), (6,5), (7,1)]$	$V(6) = [(2,5), (4,2), (5,6)]$
$V(3) = [(1,3), (2,8), (4,9), (7,7)]$	$V(7) = [(2,1), (3,7), (4,4)]$
$V(4) = [(1,10), (2,12), (3,9), (6,2), (7,4)]$	

   a) Escriba, en orden de selección, las aristas seleccionadas para la solución por el algoritmo de PRIM sobre ese grafo.

   b) Escriba, en orden de selección, las aristas seleccionadas para la solución por el algoritmo de KRUSKAL sobre ese grafo.

   Solución:

   a) Si tomamos el nodo 1 como nodo de partida, entonces las aristas seleccionadas por el algoritmo de PRIM son, por este orden: $(1,3), (3,7), (7,2), (4,7), (4,6), (5,6)$

   b) Las aristas seleccionadas por el algoritmo de KRUSKAL son, por este orden: $(2,7), (4,6), (1,3), (4,7), (5,6), (3,7)$

2. Supóngase que un grafo tiene exactamente 2 aristas que tienen el mismo peso.

   a) ¿Construye el algoritmo de PRIM el mismo árbol de recubrimiento mínimo, independientemente de cuál de esas aristas sea seleccionada antes?

   b) ¿Y si consideramos el algoritmo de KRUSKAL?

   Solución:
   En ninguno de los dos casos se construye, necesariamente, el mismo árbol de recubrimiento mínimo. Depende de cómo esté implementado el orden de selección. Compruébese con el siguiente ejemplo:

$V = [V(1), V(2), V(3)]$
$V(1) = [(2,1),(3,2)]$
$V(2) = [(1,1),(3,2)]$
$V(3) = [(1,2),(2,2)]$

Ambos algoritmos pueden dar como solución la lista de aristas $[(1,2),(1,3)]$ o $[(1,2),(2,3)]$.

3. Considérese el siguiente algoritmo para calcular las componentes conexas de un grafo no dirigido $G = (N, A)$ usando la estructura de partición:

```
proc COMPONENTES_CONEXAS(G = (N, A))
  for cada nodo v ∈ N loop CREA_CONJUNTO(v) end for
  for cada arista (x, y) ∈ A loop
    etiquetaX ← BUSCAR(x)
    etiquetaY ← BUSCAR(y)
    if etiquetaX ≠ etiquetaY then
      UNIR(etiquetaX, etiquetaY)
```

Si $G = (N, A)$ tiene k componentes conexas, ¿cuántas operaciones BUSCAR se realizan? ¿Cuántas operaciones UNIR se realizan? Exprésese el resultado en términos de $n = |N|$, $a = |A|$ y k.

Solución:
El número de operaciones BUSCAR es 2 por cada arista del grafo,es decir $2a$. El número de operaciones UNIR es $n-k$, ya que comenzamos con n componentes conexas, y cada operación UNIR reduce en 1 ese número. Si al final el número es k, el número de operaciones UNIR habrá sido $n-k$.

4. Dado un grafo $G = (N, A)$ no dirigido, con pesos asociados a las aristas, el algoritmo KRUSKAL calcula un árbol de recubrimiento mínimo de $G = (N, A)$, empleando para ello la estructura partición. El método comienza ordenando el conjunto de aristas del grafo en orden creciente según sus pesos, para pasar a continuación a tratar una a una estas aristas. Analícese una variante de dicho algoritmo en la cual se emplea un montículo (heap) para almacenar las aristas, de donde se irán cogiendo una a una para ser tratadas de la misma forma que en la versión original. Escríbase el algoritmo y calcúlese su orden en el caso peor.

Solución:
Para esta variante del algoritmo KRUSKAL usamos un montículo_min en el que los valores clave para la organización del montículo son los pesos de las aristas (que denominamos $pesos(A)$), pero mantenemos asociada a cada peso p su arista correspondiente (que denominamos $p.arista$). Entonces, podemos escribir la variante del algoritmo KRUSKAL del siguiente modo:

```
func KRUSKAL_VARIANTE (G = (N, A)) return conjunto_de_aristas
  CREAR_MONTÍCULO(pesos(A))
  for cada i ∈ N loop CREA_CONJUNTO(i) end loop
```

```
ARM ← { }   {conjunto de aristas seleccionadas}
while |ARM| ≠ |N| − 1 loop
  (u, v) ← Retirar_Raíz(A).arista
  R_u ← Buscar(u)
  R_v ← Buscar(v)
  if R_u ≠ R_v then
    ARM ← ARM ∪ {(u, v)}
    Unir(u, v)
end loop
return ARM
```

Análisis:

- Crear_Montículo($pesos(A)$) es de $\Theta(a)$ siendo a el número de aristas de A.
- Inicializar los n conjuntos de la partición es de $\Theta(n)$ siendo n el número de nodos de N.
- Retirar_Raíz(A) es de $O(\lg m)$ siendo m el número de elementos que queden en el montículo.

En el caso peor, hay que realizar el mismo número de operaciones Unir, Buscar que en el algoritmo Kruskal original, es decir $O(a \lg n)$; pero además hay que rehacer el montículo (Retirar_Raíz(A)) después de cada toma de la arista de menor peso, lo que, en el caso peor, significa hacerlo a veces. Como empezamos con $a - 1$ aristas, esto resulta

$$\lg(a-1) + \lg(a-2) + \ldots + \lg 1 \in O(a \lg a)$$

Concluyendo, el algoritmo Kruskal_Variante es de $O(a \lg a) = O(a \lg n)$.

5. El algoritmo Dijkstra calcula las distancias mínimas de los caminos desde un nodo origen a cada uno de los nodos restantes, en un grafo dirigido y con pesos no negativos asociados a las aristas.

   Escriba una adaptación de ese algoritmo, preservando el orden de complejidad, de manera que:

   a) calcule, además, el número de caminos distintos de distancia mínima que hay desde el origen a cada nodo y

   b) calcule lo suficiente para reconstruir cada uno de esos caminos mínimos.

   Solución:
   Para que el algoritmo Dijkstra original, además de determinar las distancias de los caminos mínimos, calcule lo suficiente para calcular los propios caminos (secuencias de nodos), basta con añadirle otra tabla *Precedentes*(1..n) en la que, para cada nodo x, *Precedentes*(x) registre la lista de nodos que preceden inmediatamente al nodo x en caminos de distancia mínima desde el origen a x. Por ejemplo, si *Precedentes*(x) = $[x_1, x_2, x_3]$, esto significa que (x_1, x), (x_2, x) y

(x_3, x) son últimas aristas respectivas de caminos mínimos distintos del origen a x. Suponemos que el nodo origen es el 1.

Usaremos una tabla $NumCamMin(2..n)$, de números naturales, para registrar el número de caminos mínimos del nodo 1 a cada nodo $i = 2..n$.

```
func DIJKSTRA_CON_CAMINOS (P(1..n, 1..n)) return tablas
  for i in 2...n loop Cand(i) = true end loop
  for i in 2...n loop
    D(i) ← P(1, i)
    if D(i) = ∞ then
      NumCamMin(i) ← 0
      Precedentes(i) ← [ ]
    else
      NumCamMin(i) ← 1
      Precedentes(i) ← [1]
  end loop
  for j in 1...n − 2 loop   {Hay que realizar n − 2 selecciones voraces}
    min ← ∞
    for i in 2...n loop
      if Cand(i) ∧ D(i) < min then
        min ← D(i)
        x ← i
    end loop
    Cand(x) ← false   {x es el nodo seleccionado}
    for i in 2...n loop
      if Cand(i) ∧ D(x) + P(x, i) < D(i) then
        D(i) ← D(x) + P(x, i)
        NumCamMin(i) ← NumCamMin(x)
        Precedentes(i) ← [x]
      elsif Cand(i) ∧ D(x) + P(x, i) = D(i) then
        NumCamMin(i) ← NumCamMin(i) + NumCamMin(x)
        Precedentes(i).añadir(x)
    end loop
  end loop
  return D(1..n), Precedentes(1..n) y NumCamMin(1..n)
```

Es fácil comprobar que DIJKSTRA_CON_CAMINOS es del mismo orden que DIJKSTRA.

6. Junto con los cursos de verano de la universidad va a celebrarse un ciclo de conferencias. Las conferencias son muchas y cada conferenciante ha decidido la hora de comienzo y de finalización de su conferencia. La dirección de los cursos de verano ha decidido reservar unas cuantas salas para celebrar exclusivamente las conferencias. Escriba y analice un algoritmo que determine cual es el número mínimo de salas que se necesitan.

Solución:

- Un criterio de selección voraz es el de seleccionar la conferencia que antes termine y asignarla a la primera sala en la que se pueda realizar.

  Significa reducir este problema al de selección de actividades, pero atendiendo a todas las actividades. Por lo que sabemos del problema de selección de actividades, este criterio maximiza el número de conferencias compatibles en cada sala y de ahí podemos deducir que el número de salas utilizadas será el mínimo.
- Otro criterio de selección puede ser el de seleccionar la conferencia que empiece antes y asignarla a la primera sala en la que se pueda realizar.

  A continuación demostramos la corrección del criterio, por inducción sobre el cardinal de conferencias seleccionadas. Sea una solución óptima que no asigna la primera sala a la actividad que más temprano comienza, supongamos que le asigna la sala $k > 1$. Entonces podemos trasladar todas las conferencias de la sala k a la sala 1 y todas las de la sala 1 a la sala k y ya tenemos una solución óptima que toma la misma opción que la nuestra para la actividad que termina más temprano.

  Para el paso de inducción, supongamos que una solución óptima es igual a la de nuestro criterio hasta la conferencia número $c - 1 > 1$ y que se diferencia en la opción de la conferencia que nuestro criterio seleccionaría en la iteración c. Supongamos que esa solución óptima asigna a nuestra conferencia número c una sala número k, posterior a la que elegiría nuestro criterio (obsérvese que por definición no podría ser anterior). Entonces podemos trasladar las conferencias de la sala k, desde nuestra c en adelante, a la sala donde habría puesto nuestro criterio a la c y, en correspondencia, trasladamos todas las que tenga la solución óptima en ese lugar —diferentes a nuestra selección de $c-1$ conferencias— a la sala k. Obsérvese que no hay problemas de compatibilidad por la definición de c.

7. Considere que tenemos n programas para grabar en un disco, pero el espacio de memoria que necesitan excede la capacidad del disco. Cada programa P_i requiere m_i kilobytes de memoria, la capacidad del disco es de C kilobytes y $C < \sum_{i=1}^{n} m_i$.

   a) Queremos grabar en el disco el máximo número posible de esos programas. Demuestre si es correcto el siguiente criterio de selección voraz: seleccionar el programa con menor requerimiento de memoria. Si cree que es incorrecto muestre un contraejemplo.

   b) Se plantea otro problema distinto, si lo que queremos es utilizar la máxima capacidad posible del disco en la grabación de esos programas. Demuestre si para resolver este problema es correcto el criterio de selección voraz siguiente: seleccionar el programa con mayor requerimiento de memoria. Si cree que es incorrecto muestre un contraejemplo.

Solución:

a) Demostraremos la corrección del criterio: seleccionar el programa con menor requerimiento de memoria, por inducción sobre el cardinal del conjunto de programas seleccionados.
Supongamos ordenados los programas, según requerimiento creciente de memoria: $m_1 \leq \ldots \leq m_n$.
Base de la inducción. Sea *cardinal*(S)=1, es decir que hemos seleccionado sólo el programa con requerimiento de memoria m_1.
Veamos que siempre existe una solución optimal que incluye al programa $\{1\}$ (es decir, S es prometedor). Si OPT es una solución optimal cualquiera y k es el programa más pequeño de OPT (o sea $m_k = \text{mín}\{m_j : j \in OPT\}$), pueden ocurrir dos casos: $k = 1$ o $k \neq 1$.
En el caso de $k = 1$ terminamos, puesto que eso significaría que $S \subseteq OPT$, que es lo que queremos demostrar. Si $k \neq 1$ obsérvese que $(OPT - \{k\}) \cup \{1\}$ es también una solución optimal ya que tiene el mismo número de programas que OPT, y además caben en el disco puesto que $m_1 \leq m_k$.
Hipótesis de inducción: Sea *cardinal*$(S) = r - 1$ y S es prometedor y construido según el criterio de selección indicado.
Sea OPT una solución optimal cualquiera, que incluya a $S = \{1, \ldots, r-1\}$ (nótese que debe existir una solución así por la hipótesis de que S es prometedor). Entonces podemos representar a OPT así:
$OPT = \{1, \ldots, r-1\} \cup \{p_r, \ldots, p_t\}$.
Sea p_r el programa de $OPT - S$ con menor requerimiento de memoria (o sea, $m_{p_r} = \text{mín}\{m_j : j \in OPT - S\}$). Por definición del criterio de selección, $m_r \leq m_{p_r}$. Por lo tanto, si sustituimos en OPT al programa p_r por el programa r tenemos otra solución OPT' tan buena como OPT (con el mismo número de programas); y concluimos que la selección voraz r-ésima nos lleva a un conjunto prometedor.

b) En este caso, el criterio de selección no lleva siempre a una solución óptima. Como contraejemplo puede servir el siguiente: Supongamos que $C = 13$ y que tenemos 4 programas con requerimientos de memoria 7, 5, 4 y 2, respectivamente.
Entonces el criterio de selección indicado tomaría los programas de 7 y 5; y ya no tendría espacio para más, mientras que existe una solución mejor, que consiste en elegir los programas de 7, 4 y 2.

8. Considere una red de n ordenadores personales que comparte una sola impresora. En un momento dado, las n personas que están trabajando en los ordenadores solicitan simultáneamente la impresión de un documento y se acercan a la impresora para recogerlo.

   Diseñe un algoritmo para que la impresora decida el orden en que se deben imprimir los documentos para que la suma de tiempos de espera de las n personas resulte mínimo. El tamaño de cada documento es conocido y la impresora imprime cada documento en un tiempo t_i $i = 1, \ldots, n$.

Solución:
Un criterio de selección que garantiza que la suma total de tiempos de espera es mínima, consiste en seleccionar para impresión el documento que menos tiempo necesite, de los que todavía no han sido impresos. Es decir, podemos ordenar los documentos en base a su t_i. Supongamos, sin pérdida de generalidad, que $t_1 \leq \ldots t_n$. Y se imprimen en ese orden.

La demostración de corrección de ese criterio puede hacerse por inducción sobre el número de documentos seleccionados. Vamos a nombrar a cada documento i mediante su tiempo asociado t_i.

Caso base. Supongamos que hay una solución optimal OPT que imprime los documentos de manera que el primer documento impreso no es el t_1 elegido por nuestro criterio. Vamos a nombrar $\langle s_1 \leq \ldots s_n \rangle$ a la secuencia de impresión de documentos según la solución OPT. Estamos asumiendo que $s_1 \neq t_1$, así que, por definición de t_1, $t_1 \leq s_1$. Supongamos que $t_1 = s_k$, con $k \neq 1$.

La suma de tiempos de espera de OPT es la siguiente:

$$\begin{aligned} & s_1 + (s_1 + s_2) + (s_1 + s_2 + s_3) + \ldots (s_1 + \ldots + s_n) \\ = \quad & ns_1 + (n-1)s_2 + \ldots + 2s_{n-1} + s_n \end{aligned}$$

Construimos otra solución OPT' intercambiando sólo el orden de los documentos s_1 y s_k. Vamos a demostrar que la suma de los tiempos de espera de OPT' es menor o igual que la de OPT, con lo cual justificamos que nuestra propuesta de selección para la impresión del primer documento puede llevar a una solución óptima.

La única diferencia entre la suma de tiempos de OPT' y OPT está en los sumandos correspondientes a las posiciones 1 y k. Justificaremos la relación de orden que nos interesa, siguiendo una cadena de equivalencias. Obsérvese que, mirando a OPT', $ns_k + (n-k)s_1 = nt_1 + (n-k)s_1$, dado que $(t_1 = s_k)$. Entonces:

$$\begin{aligned} & nt_1 + (n-k)s_1 \leq ns_1 + (n-k)s_k = ns_1 + (nt_1 - ks_k) \\ \Leftrightarrow \quad & -ks_1 \leq -ks_k \\ \Leftrightarrow \quad & -ks_1 \leq -kt_1 \\ \Leftrightarrow \quad & s_1 \geq t_1 \quad \text{que es verdadero, por definición de } t_1 \end{aligned}$$

El caso inductivo se reduce al caso base, ya que, una vez seleccionado el primer documento t_1, estamos ante el mismo problema inicial pero con un documento menos, y podemos repetir la misma justificación.

Por lo tanto el problema se resuelve ordenando el conjunto de documentos en orden creciente de tiempos de impresión.