

Ejercicio 1. Algoritmos voraces

[ITIS/ITIG Feb 2007] En un lejano país los estudios de neuromagnetismo capilar están organizados de la siguiente manera: existen N asignaturas, y cada asignatura proporciona una cualificación y un título, es decir, no es necesario aprobar todas las asignaturas.

La relación entre las asignaturas es la siguiente: para dos asignaturas, A y B , puede existir una relación de suficiencia de A hacia B si para matricularse en B es suficiente haber aprobado A ; o bien, pueden ser independientes: el aprobar A no da derecho a matricularse en B . Son conocidas las relaciones de suficiencia entre las asignaturas. Diseña un algoritmo, total o parcialmente voraz, que determine el mínimo número de años necesario para aprobar una asignatura dada, detallando lo siguiente:

- Las estructuras necesarias para representar la información. (0,5 puntos)
- La función o procedimiento que implemente este algoritmo. (2,5 puntos).

Solución Ejercicio 1. Algoritmos voraces

- Estructuras de datos: Como argumentos de entrada/salida del algoritmo:
 - ▶ **A[1..N,1..N]**: con valores booleanos: $A[i,j]=cierto$ si la asignatura i es suficiente para la asignatura j
 - ▶ **libre[1..N]**: con $libre[k]=cierto$ se indica que la asignatura k no necesita de ninguna otra asignatura para cursarla
 - ▶ **asig**: código de la asignatura que se quiere cursar
 - ▶ **años**: número de años necesarios para cursar asig
- Estructuras internas del algoritmo (para aplicar Dijkstra):
 - ▶ $L[1..N,1..N]$, con $L[j,i]=1$ si i es suficiente para j , $L[j,i]=\infty$ en caso contrario
 - ▶ $D[1..N]$ que contiene los caminos mínimos desde cualquier nodo hasta asig

Solución Ejercicio 1. Algoritmos voraces (cont.)

```
proc neuromag(A[1..N,1..N], libre[1..N],asig,años)
  si libre[asig] entonces años  $\leftarrow$  1
  si no
    desde i $\leftarrow$  1 hasta N hacer
      desde j $\leftarrow$  1 hasta N hacer
        si A[i,j] entonces
          L[j,i]  $\leftarrow$  1
        si no
          L[j,i]  $\leftarrow$   $\infty$ 
        fin si
      fin desde
    fin desde
  crear D[1..N] ; crear L[1..N,1..N]
  dijkstra2(L,D,asig); min  $\leftarrow$   $\infty$ 
  desde i $\leftarrow$  1 hasta n hacer
    si D[i]<min Y libre[i] entonces min  $\leftarrow$  D[i]
  fin desde
  años  $\leftarrow$  min + 1
fin si
fin proc
```

Solución Ejercicio 1. Algoritmos voraces (cont.)

```
proc dijkstra2(A[1..n,1..n], D[1..n],ini)
  C ← {1, ..., n} \ {ini}
  desde i ← 1 hasta n hacer
    D[i] ← A[ini,i]
  fin desde
  desde k ← 1 hasta n-2 hacer
    min ← ∞
    para todo i ∈ C hacer
      si D[i] < min entonces min ← D[i] ; prox ← i
    fin desde
    si min = ∞ entonces terminar // Puede haber nodos no accesibles
    C ← C \ {prox}
    para todo i ∈ C hacer
      si D[prox]+A[prox,i] < D[i] entonces
        D[i] ← D[prox]+A[prox,i]
      fin si
    fin desde
  fin desde
fin proc
```

Ejercicio 2. Algoritmos voraces

[ITIS/ITIG Jun 2007] Consideremos un pueblo en el que conocemos completamente su mapa: conocemos todas las intersecciones de las calles y todas sus longitudes. Supongamos que en cada intersección de calles existe una plaza. El alcalde de este pueblo ha decidido asfaltar las calles. Sin embargo, como el presupuesto es reducido ha decidido asfaltar aquellos tramos de calle de tal forma que todas las plazas queden unidas por tramos de calle asfaltadas. Determina qué tramos de calle son necesarios asfaltar para resolver el problema con un coste mínimo. Nota: Podemos suponer que el gasto por unidad de longitud es g . Detalla lo siguiente:

- Las estructuras y/o variables necesarios para representar la información del problema (1 punto).
- Justifica de manera clara y concisa la metodología utilizada (1 punto).
- El procedimiento o función que implemente el algoritmo (3 puntos).

Solución Ejercicio 2. Algoritmos voraces

- Este problema consiste en obtener el árbol de recubrimiento mínimo del grafo formado por todas las calles (aristas) y plazas (nodos) del pueblo.
- Como todas las calles tienen el mismo coste de asfaltado (g por cada unidad de longitud), se puede considerar directamente su longitud como el peso de cada arista
Si cada calle tuviera un coste diferente por unidad de longitud, bastaría con obtener el coste de la calle ($g_i * l_i$, donde l_i es su longitud) para saber el peso de la arista
- Las estructuras de datos y variables para representar la información dependen del algoritmo que se utilice para obtener la solución.

Solución Ejercicio 2. Algoritmos voraces (cont.)

- Argumentos del algoritmo:
 - ▶ entrada: Depende del algoritmo utilizado. En el caso del algoritmo de Prim, es más conveniente utilizar la matriz de adyacencia. Sin embargo, vamos a suponer que recibimos $D[1..M]$, con los tramos de calle identificados por las plazas de origen y destino y su longitud. Podemos suponer que las plazas están numeradas de 1 a N (N es otro argumento del algoritmo)
 - ▶ salida: conjunto F de tramos de calles a asfaltar
- Estructuras internas: Si se utiliza el método de Prim:
 - ▶ matriz de adyacencia $A[1..N][1..N]$ con las distancias entre plazas
 - ▶ $mas_cercano[i]$: índice del nodo ya considerado más cercano a i
 - ▶ $distancia[i]$: distancia de esta arista
- Si se utiliza el algoritmo de Kruskal, no son necesarias estas estructuras, pero en su lugar debe utilizarse un TAD para representar los conjuntos disjuntos

Solución Ejercicio 2. Algoritmos voraces (cont.)

- Metodología utilizada: método voraz
 - ▶ Candidatos: calles a asfaltar
 - ▶ Función solución: determinar si las calles seleccionadas unen todas las plazas con mínimo coste (árbol de recubrimiento mínimo)
 - ▶ Función de factibilidad: no debe existir ningún ciclo entre las calles seleccionadas
 - ▶ Función de selección: depende del algoritmo; si es el algoritmo de Prim, selecciona la plaza más próxima a las plazas ya consideradas
- Procedimiento que implementa el algoritmo: por ejemplo, el algoritmo de Prim (hay que incluirlo en la solución)

Solución Ejercicio 2. Algoritmos voraces (cont.)

- Si se utiliza el algoritmo de Prim, hay que hacer un algoritmo que prepare la matriz de adyacencia y lo llame:

```
proc asfalto(D[1..M], N, F)
  crear A[1..N,1..N]
  desde i ← 1 hasta N hacer
    A[i,i] ← 0
    desde j ← 1 hasta N hacer
      si i ≠ j entonces A[i,j] ← ∞
    fin desde
  fin desde
  desde k ← 1 hasta M hacer
    A[D[k].inicio,D[k].fin] = D[k].longitud
    A[D[k].fin,D[k].inicio] = D[k].longitud
  fin desde
  prim(A,F) //debe incluirse el pseudocódigo de este procedimiento
fin proc
```

- Si se utiliza el algoritmo de Kruskal, ¿cómo sería el procedimiento anterior?

Ejercicio 3. Algoritmos voraces

[GV00], p. 173 Un camionero conduce desde Bilbao a Málaga siguiendo una ruta dada y llevando un camión que le permite, con el tanque de gasolina lleno, recorrer n kilómetros sin parar. El camionero dispone de un mapa de carreteras que le indica las distancias entre las gasolineras que hay en su ruta. Como va con prisa, el camionero desea parar a repostar el menor número de veces posible.

Deseamos diseñar un algoritmo voraz para determinar en qué gasolineras tiene que parar y demostrar que el algoritmo encuentra siempre la solución óptima.

Solución Ejercicio 3. Algoritmos voraces

- La estrategia voraz de este problema consiste en recorrer el máximo número de kilómetros sin repostar
- Vamos a comprobar que esta estrategia obtiene la solución óptima en todos los casos:
- Sea $d[1..G - 1]$ un vector con las distancias entre las G gasolineras del recorrido: $d[i]$ contiene la distancia entre la gasolinera $i - 1$ y la gasolinera i (suponemos que para todo i , $d[i] < n$)
- Sea $D[i]$ la distancia desde el origen hasta la i -ésima gasolinera:
$$D[i] = \sum_{k=1}^i d[k]$$
- Sea x_1, \dots, x_s el conjunto de gasolineras obtenido por esta estrategia, y sea y_1, \dots, y_t otro conjunto de gasolineras

Solución Ejercicio 3. Algoritmos voraces (cont.)

- Debemos demostrar que $s \leq t$. Para ello, basta probar que $x_k \geq y_k$ para todo k (1)
- Sea k el primer índice tal que $x_k \neq y_k$. Si esto ocurre, por la estrategia elegida, $x_k \geq y_k$
- Además, se cumple que $x_{k+1} \geq y_{k+1}$:
 - ▶ Supongamos que $x_{k+1} < y_{k+1}$ (2)
 - ▶ Se cumple que $D[y_{k+1}] - D[y_k] < n$, pues el camión puede llegar de y_k a y_{k+1} sin repostar
 - ▶ También se cumple que $D[y_{k+1}] - D[x_k] < n$, pues $D[y_k] < D[x_k]$
 - ▶ Pero esto quiere decir que la estrategia del algoritmo habría seleccionado $x_{k+1} = y_{k+1}$, lo que contradice (2)
- Repitiendo este proceso, hemos comprobado que (1) es cierto

Solución Ejercicio 3. Algoritmos voraces (cont.)

- El algoritmo resultante es:

```
proc gasolineras(n,D[1..G-1], sol[1..G-1])  
  desde i ← 1 hasta G-1 hacer  
    sol[i] ← falso  
  fin desde  
  i ← 0 ; km ← 0  
  repetir  
    repetir  
      i ← i + 1  
      km ← km + D[i]  
    hasta km > n O i = G-1  
    si km > n entonces  
      i ← i - 1  
      sol[i] ← cierto  
      km ← 0  
    fin si  
  hasta i = G-1  
fin proc
```

Ejercicio 4. Algoritmos voraces

[GV00],[MOV04] Supongamos que disponemos de n ficheros f_1, f_2, \dots, f_n con tamaños l_1, l_2, \dots, l_n y un disquete de capacidad $d < l_1 + l_2 + \dots + l_n$.

- a) Queremos maximizar el número de ficheros que ha de contener el disquete, y para eso ordenamos los ficheros por orden creciente de su tamaño y vamos metiendo ficheros en el disco hasta que no podamos meter más. Determinar si este algoritmo ávido encuentra solución óptima en todos los casos.
- b) Queremos llenar el disquete tanto como podamos, y para eso ordenamos los ficheros por orden decreciente de su tamaño, y vamos metiendo ficheros en el disco hasta que no podamos meter más. Determinar si este algoritmo ávido encuentra solución óptima en todos los casos.

Solución Ejercicio 4. Algoritmos voraces

- a) Sea L la función que devuelve el tamaño de un archivo, y que los ficheros están ordenados en orden creciente de tamaño:

$$L(f_1) \leq L(f_2) \leq \dots \leq L(f_n)$$

- Sea m el número máximo de ficheros que caben en el disquete:

$$\sum_{i=1}^m L(f_i) \leq d < \sum_{i=1}^{m+1} L(f_i) \quad (1)$$

- Sea g_1, \dots, g_s otro subconjunto de ficheros que caben también en el disquete,

$$\sum_{i=1}^s L(g_i) \leq d \quad (2)$$

Suponemos que los g_i están ordenados en orden creciente de tamaño

- Debemos demostrar que $s \leq m$
- Sea k el primer índice tal que $f_k \neq g_k$. Por la construcción del algoritmo, $L(f_k) \leq L(g_k)$

Solución Ejercicio 4. Algoritmos voraces (cont.)

- g_k corresponde a un fichero f_a según la ordenación del algoritmo, donde $a > k$
- Del mismo modo, g_{k+1} corresponde a un fichero f_b , donde $b > a > k$, y por tanto $b > k + 1$. Por tanto, $L(g_{k+1}) \geq L(f_{k+1})$. Repitiendo este proceso:

$$L(g_i) \geq L(f_i) \quad (k \leq i \leq s)$$

- Pero esto es lo mismo que decir que

$$\sum_{i=1}^s L(g_i) \geq \sum_{i=1}^s L(f_i)$$

y si nos fijamos en la expresión izquierda, antes habíamos visto en (2) que $d \geq \sum_{i=1}^s L(g_i)$

- Por tanto, $d \geq \sum_{i=1}^s L(f_i)$. Por (1), se debe cumplir que $s < m + 1$, **por lo que $s \leq m$**

Solución Ejercicio 4. Algoritmos voraces (cont.)

- b) Este algoritmo no funciona en todos los casos. Un posible contraejemplo es el que se indica en [GV00]: si tenemos cuatro ficheros con tamaños $(15, 10, 10, 2)$, y el tamaño del disco es 22. La estrategia voraz elegiría los ficheros f_1 y f_4 , con un tamaño total $15 + 2 = 17$. Sin embargo, hay una elección mejor: f_2, f_3 y f_4 , que llenan completamente el disco.

Ejercicio 5. Algoritmos voraces

[MOV04] La Universidad tiene que planificar un evento cultural que consiste en n conferencias. Para cada conferencia se conoce la hora de comienzo y la de finalización fijadas por los ponentes. Se ha pedido al Departamento de Informática que planifique las n conferencias distribuyéndolas entre las distintas salas disponibles, de forma que no haya dos conferencias en una misma sala al mismo tiempo. El objetivo es minimizar el número de salas utilizadas, para así causar el menor trastorno al resto de las actividades académicas.

Solución Ejercicio 5. Algoritmos voraces

- De cada conferencia sabemos las horas de comienzo y finalización, que abarcan el intervalo $[c_i, f_i)$
- Una estrategia voraz consiste en considerar las conferencias por orden creciente de tiempo de comienzo
- Se recorren las conferencias ordenadas, y para cada una se intenta asignar en una sala utilizada previamente y que en ese momento no esté ocupada
- Si esto no es posible, se le asigna una sala nueva
- El número de salas necesario viene determinado por las salas necesarias en el momento de mayor demanda
- Por cada conferencia, lo único que se exige es que no se solape en la misma sala con ninguna otra conferencia

Solución Ejercicio 5. Algoritmos voraces (cont.)

- Vamos a demostrar que esta solución es óptima:
- Utilizaremos el método de inducción sobre el número de conferencias
- **Caso base:** para $n = 1$, se necesita una sala
- **Hipótesis:** supongamos que la estrategia es óptima para $n - 1$ conferencias y k salas
- **Paso de inducción:** Consideramos la conferencia n ésima:
 - ▶ Esta conferencia está definida en el intervalo $[c_n, f_n)$
 - ▶ Si esta conferencia se puede planificar en alguna de las k salas, la solución es óptima
 - ▶ Si no, podemos comprobar que la solución también es óptima:
 - ★ Si la conferencia n es incompatible con las k salas, es porque en cada sala existe una conferencia ya planificada que solapa con n
 - ★ Además, ninguna conferencia ya planificada empieza más tarde que n , y existen
$$[c_{i_1}, f_{i_1}), \dots, [c_{i_k}, f_{i_k})$$
 tales que $c_{i_j} \leq c_n < f_{i_j}$ ($1 \leq j \leq k$)
 - ★ Por tanto, existen $k + 1$ conferencias que se solapan en el tiempo a partir de c_n , por lo que son necesarias $k + 1$ salas

Solución Ejercicio 5. Algoritmos voraces (cont.)

```
//C y F son las horas de comienzo y fin de las conferencias
//sala[i] contiene la sala (de 1 a k) en la que se realiza la conferencia i
proc conferencias(C[1..N], F[1..N], sala[1..N], k)
  //ocupacion[j] almacena la hora hasta la que está ocupada la sala j
  crear ocupacion[1..N]
  ordenar(C,F) // Debe implementarse este procedimiento
  k ← 0
  desde i ← 1 hasta N hacer
    j ← 1
    mientras j ≤ k Y ocupacion[j] > C[i] hacer
      j ← j + 1
    fin mientras
    si j ≤ k entonces
      sala[i] ← j ; ocupacion[j] ← F[i]
    si no
      k ← k + 1
      sala[i] ← k ; ocupacion[k] ← F[i]
    fin si
  fin desde
fin proc
```

Ejercicio 6. Algoritmos voraces

[NN98], [BB97] **Minimización del tiempo total en un sistema.** Tenemos una serie de procesos a ejecutar en un ordenador, que ejecuta los procesos en orden secuencial. Queremos minimizar el tiempo total que los procesos permanecen en el sistema: la suma de los tiempos de espera más los tiempos de ejecución.

Suponemos que los tiempos que va a tardar la ejecución de cada uno de los procesos son conocidos. Se pide definir una estrategia voraz que determine en qué orden ejecutar los procesos y demostrar que es óptima.

Solución Ejercicio 6. Algoritmos voraces

- Una solución sencilla consiste en considerar todas las planificaciones posibles y tomar el mínimo
- La complejidad de un algoritmo de este tipo es $\mathcal{O}(n!)$
- Veamos un ejemplo: con tres procesos, $t_1 = 5$, $t_2 = 10$, y $t_3 = 3$, tenemos los siguientes casos:

Orden	Tiempo total
1 2 3:	$5+(5+10)+(5+10+3)=38$
1 3 2:	$5+(5+3)+(5+3+10)=31$
2 1 3:	$10+(10+5)+(10+5+3)=43$
2 3 1:	$10+(10+3)+(10+3+5)=41$
3 1 2:	$3+(3+5)+(3+5+10)=29$
3 2 1:	$3+(3+10)+(3+10+5)=34$

- Parece que la planificación óptima es la que va ejecutando los procesos en orden no decreciente de duración

Solución Ejercicio 6. Algoritmos voraces (cont.)

- Debemos demostrar formalmente que esta estrategia es óptima
- Sea $P = p_1, p_2, \dots, p_n$ una planificación óptima, que minimiza el tiempo total T en el sistema
- Dado p_j , $1 \leq j \leq n$, sea T_j el tiempo total de p_j : la suma del tiempo de espera de p_j más el tiempo de ejecución: $T_j = \sum_{k=1}^j t_k$
- El tiempo total en el sistema es por tanto $T = \sum_{j=1}^n T_j$
- Hay que demostrar que los procesos están en orden no decreciente, $t_1 \leq t_2 \leq \dots \leq t_n$ (t_i son las duraciones de la ejecución de los procesos de la planificación)
- Utilizamos la técnica de **reducción al absurdo**: Supongamos que no están en orden no decreciente: al menos existe un i , $1 \leq i < n$, tal que $t_i > t_{i+1}$

Solución Ejercicio 6. Algoritmos voraces (cont.)

- Podemos intercambiar ambos trabajos, creando una nueva planificación $P' = p_1, \dots, p_{i-1}, p_{i+1}, p_i, p_{i+2}, \dots, p_n$
- En esta nueva planificación, sólo se ven afectados los tiempos de los procesos t_i y t_{i+1}
- En la nueva planificación P' , los tiempos totales de p_i y de p_{i+1} son ahora distintos:

$$T'_i = T_i + t_{i+1}, \text{ y } T'_{i+1} = T_{i+1} - t_i$$

- Entonces: $T' = T + t_{i+1} - t_i$
- Como $t_{i+1} < t_i$, $T' < T$, con lo que llegamos a una contradicción, pues la planificación P no es óptima ya que existe otra, P' , con un tiempo total menor
- Por tanto, la suposición realizada es falsa: **para que la planificación P sea óptima los procesos deben estar en orden no decreciente**

Ejercicio 7. Algoritmos voraces

[ITIS/ITIG Feb 2008] Las emisiones de los automóviles privados son una de las principales fuentes de contaminación en las grandes ciudades. Las autoridades universitarias han decidido contribuir seriamente a la mejora de la calidad del aire prohibiendo el paso de automóviles por el campus y creando una serie de líneas de autobuses eléctricos que permitan acceder a distintos puntos de la Ciudad Universitaria.

Para ello se han estudiado los principales flujos de vehículos entre distintos puntos seleccionados (facultades, estación de metro, etc.) de la Ciudad Universitaria. Los resultados del estudio contienen el número de vehículos que transitan al día entre estos puntos seleccionados (sin tener en cuenta el sentido en el que circulan). El objetivo de este estudio es localizar los trayectos entre puntos seleccionados que utilizan más vehículos y que permiten conectar todos los puntos seleccionados.

Diseña mediante el **método voraz** un algoritmo que proporcione estos trayectos. Demuestra que la solución que propones es óptima (Utiliza los teoremas vistos en clase si lo crees necesario)

Solución Ejercicio 7. Algoritmos voraces

- Este problema es similar al árbol de recubrimiento mínimo de un grafo: los puntos seleccionados son los nodos del grafo, y los trayectos son las aristas
- La diferencia está en que nos piden el conjunto de aristas por las que pasan más vehículos y que nos permitan conectar todos los puntos seleccionados: es el **árbol de recubrimiento máximo**
- En la información que se proporciona, las aristas están etiquetadas con el número de vehículos que pasan: si no hay vehículos que circulen entre dos puntos (sin pasar por ningún otro punto intermedio), la arista tendrá un 0
- Una forma posible de resolverlo es transformando la información de entrada de forma que, calculando el árbol de recubrimiento mínimo del grafo transformado, se obtenga el mismo árbol que el de recubrimiento máximo
 - ▶ Por ejemplo, se puede hacer $L'[i, j] = 1/L[i, j]$
- También se puede modificar cualquiera de los algoritmos vistos para el caso mínimo

Solución Ejercicio 7. Algoritmos voraces

```
proc recmax(A[1..N,1..N], F)
  crear mas_cercano[2..N], distancia[2..N]
   $F \leftarrow \emptyset$ 
  desde j  $\leftarrow$  2 hasta N hacer
    mas_cercano[j]  $\leftarrow$  1 ; distancia[j]  $\leftarrow$  A[j,1]
  fin desde
  desde i  $\leftarrow$  1 hasta N-1 hacer
    max  $\leftarrow$   $-\infty$ 
    desde j  $\leftarrow$  2 hasta N hacer
      si  $0 \leq$  distancia[j]  $>$  max entonces max  $\leftarrow$  distancia[j] ; prox  $\leftarrow$  j
    fin desde
    F  $\leftarrow$  F  $\cup$  {(mas_cercano[prox],prox)}
    distancia[prox]  $\leftarrow$  -1
    desde j  $\leftarrow$  2 hasta N hacer
      si A[j,prox]  $>$  distancia[j] Y distancia[j]  $\neq$  -1 entonces
        distancia[j]  $\leftarrow$  A[j,prox] ; mas_cercano[j]  $\leftarrow$  prox
      fin si
    fin desde
  fin desde
fin proc
```

Solución Ejercicio 7. Algoritmos voraces

- También se podía haber utilizado el algoritmo de Kruskal.
- Este algoritmo es muy parecido que el algoritmo de Prim para recubrimiento mínimo.
- Para demostrar que este algoritmo es óptimo, podemos utilizar el resultado que ya conocemos de optimalidad del algoritmo de Prim (o Kruskal).
- Si el algoritmo de Prim lo aplicamos sobre un grafo G' resultado de invertir el peso de todas las aristas del grafo original G , se comportará exactamente como el algoritmo de este problema.
- Por tanto, el árbol de recubrimiento mínimo de G' se corresponderá con el árbol de recubrimiento generado por este algoritmo (a excepción de los valores de las aristas)
- En el grafo original G , este árbol se corresponde con el árbol de recubrimiento máximo
- Por tanto, este algoritmo nos proporciona el **árbol de recubrimiento máximo de G**