Control parcial de Esquemas Algorítmicos (IG24)

16 de abril de 2005

1. Estructuras de datos — 1,5 puntos

Deseamos añadir una nueva operación a las disponibles sobre los diccionarios de prioridad: la eliminación de un elemento dada su clave.

Describe brevemente como se podría realizar dicha operación de la forma más eficiente posible, indicando y justificando el coste temporal que presentaría la operación. Acompaña la descripción con un ejemplo gráfico de la estructura completa que, como mínimo, la muestre antes y después de efectuar la operación.

2. Voraces — 4,25 puntos

Un grupo de n amigos se reúnen de tanto en tanto para correrse una juerga. Cuando a uno le apetece, convoca la reunión enviando un mensaje de móvil a todos los demás, en el que indica lugar y hora (sin esperar respuesta, al que le viene bien acude y al que no, no lo hace). Todos los amigos tienen teléfono móvil y saben el número de cada uno de los otros. Se conoce el coste $c(i, j)$ de que un amigo i envie un mensaje a otro j, para todo par de amigos, $i, j \in \{1, 2, \ldots, n\}$. Todos los amigos han suscrito un nuevo tipo de plan intercompañías, "amistad dos a dos", que permite que, aunque el precio que paga cada uno por enviar un mensaje a cualquier otro sea dependiente de la compañia y su contrato particular, cueste lo mismo enviar un mensaje a otra persona suscrita que que ella te lo envie a tí ($c(i, j) = c(j, i)$). Un ejemplo de coste de mensajes para $n = 5$ amigos puede verse en la tabla inferior izquierda que, como se puede comprobar y debido al plan, es equivalente a la que se muestra a su derecha:

c(i,j)	1	2	3	4	5
1	-	15	20	10	25
2	15	-	25	5	30
3	20	25	-	15	20
4	10	5	15	-	20
5	25	30	20	20	-

$\Leftrightarrow$

c(i,j)	1	2	3	4	5
1	-	15	20	10	25
2		-	25	5	30
3			-	15	20
4				-	20
5					-

Los amigos desean modificar el sistema utilizado para quedar (el que convoca manda un mensaje a todos los demás) estableciendo un sistema de reenvío del mensaje de convocatoria con el fin de que el coste total del envío de mensajes sea el menor posible (el coste total es la suma de los costes de todos los mensajes enviados). Para ello, van a establecer una *red de amigos conectados*, que será un conjunto de $n - 1$ pares de valores (i, j) que permita conectar a todos los amigos, es decir, que si un amigo recibe un mensaje de otro que está conectado con él a través de la red, lo reenvíe a los restantes que también aparezcan unidos a él en la red. El problema consiste en encontrar, de todas las posibles combinaciones de amigos conectados, aquella que tiene un mínimo coste.

En el ejemplo anterior, la combinación $\{(1, 3), (3, 2), (4, 3), (5, 4)\}$ es una *red de amigos conectados*: si, p.ej., el amigo 1 decide enviar una convocatoria, se la enviará al 3. Al recibir éste el mensaje, se lo reenviará al 2 y al 4. De éstos, tan sólo el 4 lo reenviará, en este caso al 5. El coste de enviar un mensaje con esta red es $20 + 25 + 15 + 20 = 80$.

El problema, pues, es encontrar la mejor red. Para comenzar, se pide:

a) Se trata de un problema de optimización. Indica las características que debe tener una solución factible, cuál es la función objetivo y que debe cumplir una solución factible para ser la óptima.

b) ¿Puede haber más de una solución óptima para este problema? Sobre el ejemplo presentado, ¿cuál sería una solución óptima?

Para resolver el problema hemos diseñado las siguientes estrategias voraces:

- "La mejor conexión dos a dos": partimos de un amigo cualquiera, x, y seleccionamos al amigo al que le puede enviar un mensaje con el menor coste posible, y, e incorporamos ese par, (x, y), a la solución. A continuación, estudiamos la mejor conexión de y con cualquier otro amigo (excluyendo a x), (y, z), y la añadimos a la solución. Para el nuevo amigo z, buscamos la conexión de menor coste (excluyendo a los dos ya considerados, x e y), y así sucesivamente hasta llegar al último, que no deberá establecer ninguna conexión adicional.
- "La mejor conexión dos a dos, a partir del mejor primero": identica a la anterior, pero el amigo por el que se comienza debe ser uno de los que puede enviar un mensaje con el menor coste posible.
- "Las mejores conexiones aisladas": en este caso, seleccionamos directamente las conexiones (en primer lugar la de menor coste, luego la siguiente, y así sucesivamente), quedándonos con las n-1 de menor coste.

Se pide también:

c) Indica y justifica para cada una de las estrategias propuestas:

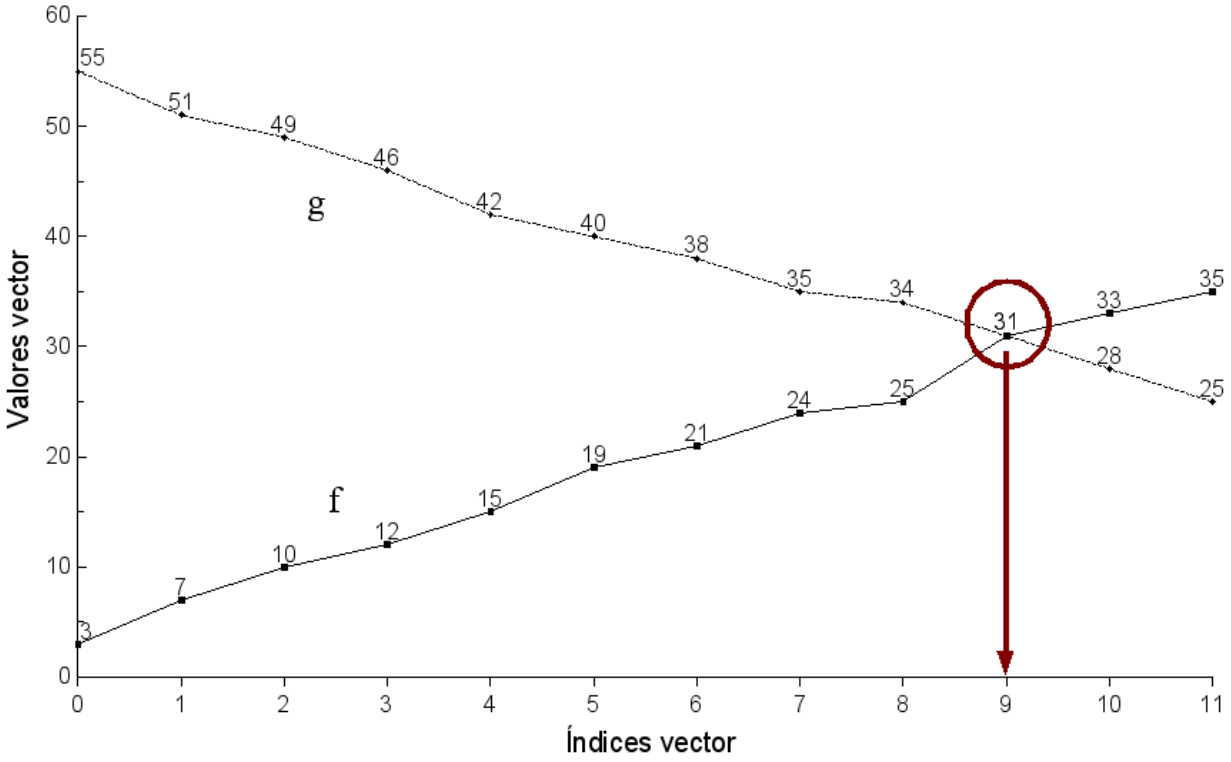
1. Si encuentra siempre una solución factible y, en caso de hacerlo, si siempre es la óptima. Debes demostrar con un contraejemplo los casos en que no esté garantizada la obtención de una solución factible y/u óptima.
2. El coste temporal de la estrategia (indica para ello las estructuras de datos concretas que utilizarías en la implentación de un algoritmo para la estrategia, describiendo los detalles del mismo que den lugar al coste señalado).

d) Si de entre las anteriores has detectado más de una estrategia que garantice la obtención de la solución óptima, dí cuál emplearías.

e) Si crees que puede haber alguna estrategia correcta alternativa a las propuestas, coméntala, señalando también su coste y las estructuras de datos empleadas (en el caso en que haya más de una estrategia posible, cita únicamente la mejor).

3. Divide y vencerás **4,25 puntos**

Sean f y g dos vectores de n elementos que representan los valores que toman dos funciones en el intervalo entero $[0 \ldots n-1]$. Los dos vectores están ordenados, aunque el primero, f, es un vector estrictamente creciente ($f[0] < f[1] < \ldots < f[n-1]$) y g es un vector estrictamente decreciente ($g[0] > g[1] > \ldots > g[n-1]$). Las curvas que representan dichos vectores se cruzan en un punto concreto, y nos interesa saber si dicho punto está contenido entre los que se nos proporcionan en ambos vectores, es decir, si existe un valor i tal que $f[i] = g[i]$ para $0 \leq i \leq n-1$.



a) Diseña un algoritmo recursivo mediante la técnica de divide y vencerás que devuelva el índice en el que coinciden los valores de las funciones o un valor especial en caso de que no exista dicho índice. El algoritmo debe ser lo más eficiente posible.

b) Indica si el algoritmo presenta un comportamiento uniforme o existe un mejor y un peor caso. Si el comportamiento es uniforme, trata de modificarlo para que ante determinadas instancias pueda finalizar antes su ejecución. Señala en qué condiciones (ante que tipos de entradas) puede el algoritmo comportarse en el mejor de los casos y en qué condiciones en el peor (da algún ejemplo de cada una de esas entradas).

c) Realiza un análisis de la complejidad temporal y espacial del algoritmo presentado, justificando los costes en el mejor y en el peor de los casos.

d) Si el algoritmo propuesto presenta recursividad por cola, elimínala e indica si alguno de los costes calculados en el apartado anterior varía.

e) ¿Se te ocurre algún algoritmo iterativo sencillo que resuelva el mismo problema sin utilizar la técnica de divide y vencerás (no es necesario que lo implementes, tan sólo descríbelo)? Indica el coste asintótico que presenta y si, atendiendo a dicho coste, puede considerarse mejor o peor que el que has diseñado.

Segundo control de Esquemas Algorítmicos (IG24)

28 de mayo de 2005

1. Programación dinámica — 4 puntos

Un joven emprendedor que acaba de ser contratado como becario en una sucursal de una gran multinacional se ha trazado como objetivo llegar a ser el presidente de la compañía en el menor tiempo posible. En la empresa hay C categorías de empleados, numeradas jerárquicamente, de manera que a los becarios les corresponde el nivel 1 y al presidente el C. El sistema de promoción interna en la empresa es muy exigente, en concreto:

- No es posible ascender a una categoría c si no se está ocupando una plaza, bien en la categoría inmediatamente inferior (la $c-1$), bien en alguna de las dos inferiores (la $c-1$ y la $c-2$), o bien en alguna de las tres inferiores ($c-1$, $c-2$ y $c-3$). Cada categoría c ($1 < c \leq C$) tiene asociado un valor a_c qué nos indica cuál es la categoría más baja desde la que se puede llegar a ella.

  Por ejemplo, si $C = 7$ y $a = [None, 1, 1, 2, 4, 3, 5]$, podemos ver que el valor de a_7 es 5, lo que indica que a la categoría 7 se permite ascender desde la 5 o la 6, que $a_6 = 3$ (a la categoría 6 se puede ascender desde la 3, la 4 o la 5), o que $a_5 = 4$ (a 5 sólo se puede ascender desde 4). El valor para la categoría 1 es $None$ puesto que no se accede desde ningún otro puesto de la empresa.
- Además de la restricción anterior, únicamente se puede ascender a una determinada categoría c desde una categoría de nivel inferior válida, $c' \in [a_c..c-1]$, si se reúnen una serie de méritos. Los méritos que hay que reunir para ascender a la categoría c dependen tanto de c como de la categoría de procedencia c'.

El becario dispone de una estimación del tiempo que le costará reunir los méritos necesarios para la transición entre cualquier par de categorias posibles $t(c', c)$.

Por tanto, en su objetivo de ascender desde becario a presidente en el menor tiempo posible, se pide:

1. Con las condiciones anteriormente expuestas:
   a) Formaliza el problema en términos de optimización,
   b) plantea la ecuación recursiva de programación dinámica que calcula el mínimo tiempo necesario, indicando cuál debe ser la llamada que se efectué a dicha función para obtener el valor óptimo, y
   c) representa el grafo de dependencias entre las llamadas del algoritmo recursivo para el ejemplo anteriormente citado, *razonando* los costes espacial y temporal con los que se puede efectuar el cálculo sobre dicho grafo (indica explícitamente, justificando tu respuesta, si es posible efectuar una reducción de complejidad espacial al realizar el cálculo iterativo).
2. La compañía está pensando en establecer un requisito adicional a los anteriores, que es que la persona que ocupe la presidencia debe contar con la experiencia de haber pasado al menos por N categorías inferiores. Realiza las modificaciones oportunas en el planteamiento del problema para responder nuevamente a las cuestiones a) y b) planteadas en el apartado 1.

2. Programación dinámica — 3,25 puntos

En una cadena de montaje, la construcción de un artefacto requiere la realización de E ensamblajes de piezas que hay que efectuar de forma secuencial. Para ello, la empresa dispone de P puestos ocupados por operarios que se disponen a lo largo de una cinta en la que se van completando los ensamblajes. En cada puesto se realizan siempre las mismas tareas. Los ensamblajes están numerados entre 1 y E y los puestos también se identifican de 1 a P, siendo el puesto 1 donde se comienza a construir el aparato y el P en el que se realiza el ensamblaje final.

El número de ensamblajes es bastante mayor que el de puestos, por lo que en cada puesto puede que tengan que realizarse varios ensamblajes. Si al puesto de índice p le asignamos los ensamblajes $i, i+1, \ldots, j$, el *disgusto* que siente el operario encargado del puesto puede estimarse mediante una función $d(i, j, p)$ (al tener que realizarse los ensamblajes de forma secuencial, los correspondientes a un puesto tendrán que ser consecutivos; el mismo razonamiento sirve para indicar que si el último ensamblaje asignado a un puesto p es el de índice j, el primero del puesto $p+1$ debe ser el $j+1$).

La empresa quiere saber cuál es la asignación de ensamblajes que debe realizar entre puestos respetando las anteriores restricciones de forma que el disgusto total de los trabajadores sea el mínimo posible. Para ello, ha aplicado la técnica de programación dinámica, llegando a la siguiente ecuación recursiva:

$$D(j,p) = \begin{cases} 0, & \text{si } j = 0 \text{ y } p = 0; \\ +\infty, & \text{si } j = 0 \text{ y } p > 0; \\ +\infty, & \text{si } j > 0 \text{ y } p = 0; \\ \min\limits_{0 \leq i < j} D(i, p-1) + d(i+1, j, p), & \text{en otro caso.} \end{cases}$$

La llamada $D(E, P)$ proporciona el valor del mínimo disgusto que se puede obtener con una asignación válida.
Se pide:

a) Representa el grafo de dependencias para la llamada $D(6,3)$. Indica qué representa un estado (j,p) y un arco entre dos estados (j',p') y (j,p) dentro del grafo.

b) Diseña un algoritmo iterativo que devuelva la *asignación óptima* de ensamblajes que le corresponde a cada puesto.

c) Indica cuál es el coste temporal y espacial del algoritmo diseñado justificando brevemente tu respuesta.

d) ¿Crees que sería posible reducir el coste espacial si únicamente nos interesará obtener el valor del mínimo disgusto? Si es así, índica cuál sería dicho coste y justifícalo.

3. Búsqueda con retroceso — 2.75 puntos

Disponemos de un conjunto A de n números enteros (tanto positivos como negativos) sin repeticiones almacenados en una lista. Dados dos valores enteros, m y C, siendo $m < n$, queremos encontrar un algoritmo de *búsqueda con retroceso* que resuelva el problema de encontrar un subconjunto de A compuesto por *exactamente* m elementos y tal que la suma de los valores de esos m elementos sea C. A continuación, se presenta el esquema de un posible algoritmo de *backtracking* que resuelve este problema.

```
def subconj_m_C(A, C, m):
  s = [None] * m

  def backtrack(i):
    if es_completa(i):
      if es_factible(): return s
    else:
      for s[i] in xrange(len(A)):
        if es_prometedora(i):
          found = backtrack(i+1)
          if found != None: return found
    return None

  return backtrack(0)
```

Si ejecutamos este programa Python con los siguientes datos de entrada: $A = [1, 3, -4, 10, 11, -3]$, $C = 10$ y $m = 3$, el resultado será:

```
Solución: [1, 2, 4]
```

que se corresponde con el siguiente desglose:

```
Elemento de L en pos. 1 es 3
Elemento de L en pos. 2 es -4
Elemento de L en pos. 4 es 11
Total: 3 + -4 + 11 = 10
```

Se pide que:

a) Escribas el código correspondiente a la función `es_completa(i)`, es decir, que definas cuál es el criterio que permite determinar si un estado es una *solución completa*.

b) Escribas el código correspondiente a la función `es_factible()`, es decir, que definas cuál es el criterio que permite determinar si una *solución completa* es una *solución factible* o no.

c) Escribas el código correspondiente a la función `es_prometedora(i)`, es decir, que definas cuál es el criterio que permite determinar si una *solución parcial* puede conducir a una solución factible (en cuyo caso hay que ramificarla) o no.

d) A la vista del código desarrollado en los apartados anteriores, ¿cuál sería la complejidad espacial del algoritmo desarrollado? ¿Y la complejidad temporal en el *mejor de los casos*?

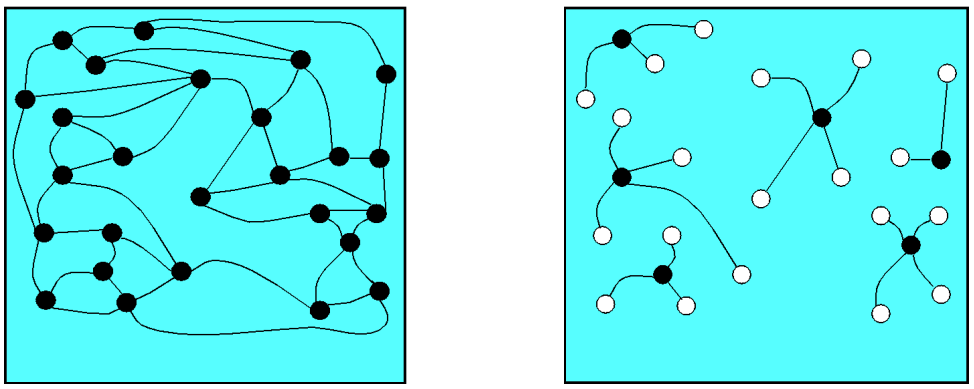
e) Realiza una traza de ejecución de este algoritmo para los valores de entrada: $A = [3, 5, -2, 4, -1]$, $C = 3$ y $m = 2$. Para ello, tendrás que dibujar el árbol de llamadas efectuadas, incluyendo solamente aquellos estados que se generan durante el proceso de búsqueda. Marca de forma especial los estados que se estudian pero se descartan por no superar la condición de ser prometedores.

Examen convocatoria ordinaria de Esquemas Algorítmicos (IG24)

17 de junio de 2005

1. Voraces **2,5 puntos**

Un *conjunto dominante* de vértices en un grafo no dirigido es un subconjunto de vértices del grafo que cumplen que, considerándolos a ellos y a todos los vértices que son adyacentes a ellos, se tienen todos los vértices del grafo. La Figura 1 ilustra este concepto:



(a) (b)

Figura 1: (a) Ejemplo de grafo no dirigido. (b) Un conjunto dominante sobre dicho grafo formado por seis vértices en el que los vértices sombreados son los que forman el conjunto dominante.

Se plantea encontrar, dado un grafo no dirigido, el conjunto dominante compuesto por *el menor número posible de vértices*. En la práctica existen *algoritmos voraces* que intentan solucionar este problema. Se pide que:

a) Ya que se trata de un problema de optimización, debes indicar las características que posee una *solución factible*, cuál es la función objetivo que se pretende optimizar y qué propiedad debe cumplir una solución factible para ser considerada óptima.

b) Para resolver este problema, se plantea la siguiente estrategia voraz: "Ordenar los vértices de mayor a menor grado en una lista L. Mientras queden elementos en L, se selecciona el primero de la lista, se añade al conjunto solución S (inicialmente vacío) y se borra dicho vértice y todos sus sucesores de L. El conjunto de vértices S resultante al final del proceso es la solución." Con respecto a esta estrategia:

   1. ¿Garantiza encontrar una solución factible? Justifica tu respuesta.
   2. ¿Encuentra siempre una solución óptima? Justifica la respuesta (si la respuesta es no, debes incluir un contraejemplo).
   3. ¿Cuál sería el resultado que obtendría si la aplicamos al grafo que aparece en la Figura 2? Justifica la respuesta indicando los diferentes pasos que efectuaría un algoritmo basado en esta estrategia.

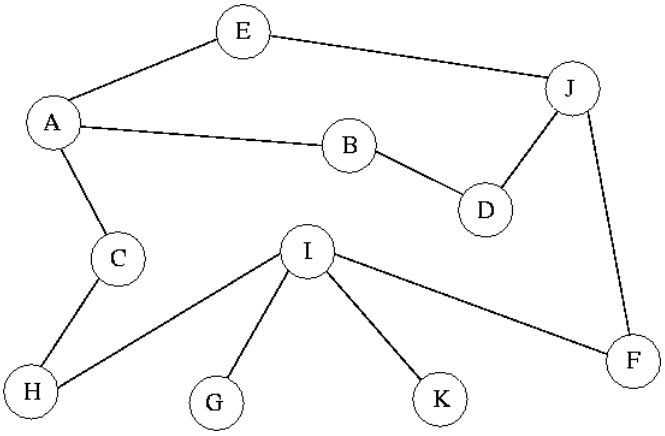


Figura 2: Ejemplo de grafo no dirigido para efectuar las trazas.

c) Planteamos ahora esta otra estrategia voraz: "Ordenar los vértices de menor a mayor grado en una lista L. Mientras queden elementos en L, se selecciona el primer vértice de la lista, v. Para todos los sucesores de este vértice v que estén en L, se coge el de mayor grado u (si no hay ningún sucesor de v en L, u es v). Se añade el vértice u al conjunto solución S (inicialmente vacío) y se borra dicho vértice y todos sus sucesores de L. El conjunto de vértices S resultante al final del proceso es la solución." Responde a las mismas preguntas (1, 2 y 3) del anterior apartado para esta nueva estrategia.

2. Divide y vencerás — 2,5 puntos

Como parte de un programa de procesamiento de textos, la empresa para la que trabajas te pide que hagas una rutina que, dada una cadena alfanumérica c y una secuencia s formada por *exactamente* dos caracteres, devuelva el valor booleano `True` si la secuencia de dos caracteres es una subcadena de la cadena alfanumérica y `False` en caso contrario. Fíjate que no importa el número de veces que la secuencia pueda encontrarse en la cadena, ni las posiciones en las que pueda estar, lo único que interesa es devolver el valor booleano correspondiente.

Por ejemplo, dada la cadena $c =$ `h72uitetei`, ante las secuencias $s =$ `ui`, $s =$ `h7`, $s =$ `ei` y $s =$ `te` devolverá `True` y ante las secuencias $s =$ `2i` y $s =$ `ee` devolverá `False`.

a) Diseña un algoritmo *recursivo* que resuelva este problema empleando la técnica de *divide y vencerás*. El algoritmo desarrollado debe presentar una complejidad espacial *no mayor* de $O(\log n)$. Recuerda que debes indicar explícitamente cuál es la llamada inicial a la función recursiva que desarrolles.

b) El algoritmo que has desarrollado en el apartado anterior, ¿tiene mejor y peor caso o se comporta de manera uniforme? Describe claramente cuál es el mejor y el peor caso. Si el algoritmo presenta un comportamiento uniforme, modifícalo para que tenga un mejor y un peor caso. Finalmente, realiza un análisis de la complejidad temporal y espacial del algoritmo justificando tu respuesta.

c) Existe un sencillo algoritmo iterativo (no desarrollado mediante divide y vencerás) que resuelve igualmente el problema planteado: descríbelo brevemente. Desde el punto de vista asintótico, ¿este algoritmo es mejor o peor que el recursivo realizado mediante divide y vencerás? Responde analizando tanto la complejidad temporal como la espacial.

3. Programación dinámica — 3,5 puntos

En un tablero de ajedrez de tamaño $n \times n$ (donde n es un número par y $n \geq 2$) se disponen una serie de *pesos* (números enteros) en las casillas de color blanco (está prohibido circular por las casillas negras). Disponemos de una ficha que podemos ubicar en cualquiera de las casillas blancas de la columna situada más a la izquierda (las casillas iniciales no tienen asignado ningún peso, es decir, se puede considerar que el peso de todas es el mismo e igual a 0). La ficha sólo puede desplazarse una casilla en cada movimiento y no puede volver atrás, es decir, sólo hay dos movimientos posibles: las dos diagonales de izquierda a derecha (hacia arriba o hacia abajo). Cada vez que la ficha pasa por una casilla obtiene como "recompensa" el peso asignado a la misma. Un *camino* sobre dicho tablero consiste en una *secuencia de movimientos que permiten a la ficha cruzarlo de izquierda a derecha*. Cada camino tiene asociado un *valor*, que es igual a la suma de los pesos asignados a las casillas recorridas por la ficha.

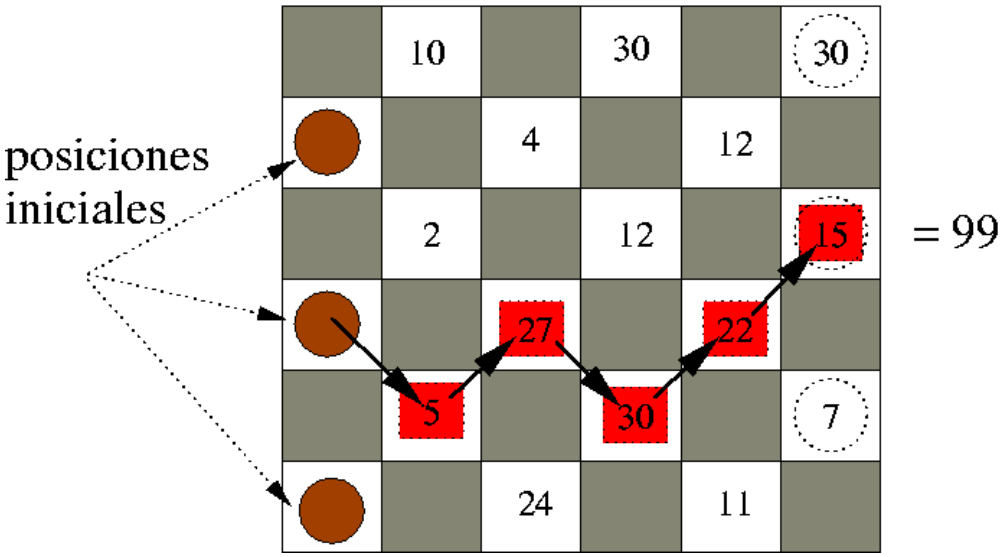


Figura 3: Camino de máximo valor en un tablero de 6×6 ($n = 6$). Las flechas indican el camino efectuado por la ficha. La posición de partida puede ser cualquiera de las casillas blancas de la columna situada más a la izquierda. La posición final puede ser cualquiera de las casillas blancas de la columna situada más a la derecha. Los cuadrados en las casillas indican la serie de recompensas obtenidas. Al final se indica la recompensa conseguida usando el camino de máximo valor.

Los datos de entrada para este problema son la dimensión del tablero n, y una función $p(c, f)$ que, dadas las coordenadas de una casilla (una columna c y una fila f), devuelve el peso almacenado en dicha casilla. Por ejemplo, para el tablero de la Figura 3, algunos valores de f son $f(2, 2) = 5$, $f(2, 6) = 10$, $f(3, 1) = 24$ o $f(6, 4) = 15$. Las casillas negras no tienen definido un valor de la función. Se pide que desarrolles un algoritmo mediante *programación dinámica* que resuelva el problema de encontrar el *valor* asociado al camino de máximo valor sobre un tablero de estas características. Para ello:

a) Presenta una formalización del problema en términos de optimización (el conjunto de soluciones factibles y la función objetivo),

b) plantea la ecuación recursiva que calcula el máximo valor posible de un camino a través del tablero en las condiciones previamente descritas, indicando cuál debe ser la llamada que se efectúe a dicha función para obtener el valor óptimo, y

c) diseña un algoritmo iterativo que devuelva dicho valor, justificando su coste temporal y espacial.

d) Suponiendo que las casillas de partida (las de la primera columna) también tuvieran pesos, indica como afectaría eso a la ecuación recursiva que has planteado y modifícala en consecuencia. Comenta si este hecho tiene incidencia en los costes asintóticos del algoritmo.

4. Búsqueda con retroceso **1,5 puntos**

Un camión cisterna cargado de gasoil sale de un almacén origen X y debe de llegar a un almacén destino Y. Por el camino, debe realizar visitas para llenar el depósito de la caldera del sistema de calefacción de una serie de chalés. El conductor de la cisterna conoce todas las carreteras que hay entre X, Y y los chalés con caldera, así como su distancia en kilómetros. Todas las carreteras son de doble sentido. Por otro lado, el camión cisterna tiene un tacógrafo electrónico que le impide efectuar recorridos de más de T km. consecutivos sin efectuar parada alguna. El programador de la empresa de suministros ha recibido el encargo de desarrollar un programa informático que permita (si es posible) efectuar el recorrido de X a Y *visitando todos los chalés una única vez* y de tal manera que las paradas que debe efectuar el camión para reiniciar la cuenta del tacógrafo coincidan con el llenado de la caldera de los chalés y así se pierda el menor tiempo posible.

El programa recibe como datos: un conjunto, V, con el número que identifica cada chalé (de 1 a n) más el número 0 que identifica el origen X y el $n+1$ que identifica el destino Y; otro conjunto, E, que indica las carreteras existentes (mediante pares) y su distancia en km.; y el número de km. consecutivos que permite recorrer el tacógrafo, T.

Por ejemplo, se han de rellenar las calderas de 5 chalés, $V = \{0, 1, 2, 3, 4, 5, 6\}$, con esta lista de carreteras: $E = \{(0, 1) : 45, (0, 3) : 80, (0, 5) : 40, (1, 3) : 100, (1, 2) : 30, (2, 4) : 50, (2, 5) : 60, (2, 6) : 90, (3, 5) : 40, (3, 6) : 55, (4, 5) : 35, (5, 6) : 70\}$ y $T = 75$. Una posible solución sería: $(0, 1, 2, 4, 5, 3, 6)$ ya que por ninguna de las carreteras empleadas se recorren más de 75 km. Sin embargo, la solución $(0, 3, 1, 2, 4, 5, 6)$ no sería factible ya que la carretera que une el almacén X con el chalé 3 tiene 80 km. y la carretera que une el chalé 3 con el 1 tiene 100 km.

El informático de la empresa está diseñando un algoritmo de *búsqueda con retroceso* para resolver este problema. Sin embargo, tiene unas cuantas dudas a la hora de desarrollarlo y como sabe que estás cursando la asignatura de *Esquemas Algorítmicos* te pide que le ayudes a resolverlas:

a) ¿Qué estructura de datos usarías para representar *de forma compacta* los *datos de entrada* del algoritmo, es decir, los chalés, los almacenes (origen y destino) y las carreteras que los unen (incluyendo su distancia en km.)? Debes indicar una única estructura de datos.

b) ¿Qué estructura o estructuras de datos utilizarías en el algoritmo para *resolver* de la manera más eficiente posible el problema?

c) ¿Cuál sería la función de *ramificación* (*branch*)? Debes indicar el (pseudo)código o bien una descripción lo más precisa posible.

d) ¿Cuál tendría que ser la implementación de la función que permite determinar si una solución es prometedora o no (*is_promising*)? Igualmente se pide el (pseudo)código o bien una descripción lo más precisa posible.

e) Teniendo en cuenta las decisiones tomadas en los apartados anteriores, el informático de la empresa quiere que le ayudes a depurar el código haciendo una traza con estos datos de entrada: $V = \{0, 1, 2, 3, 4\}$, $E = \{(0, 1) : 100, (0, 2) : 80, (0, 3) : 40, (1, 2) : 30, (1, 4) : 70, (2, 3) : 20, (3, 4) : 50\}$, $T = 70$. Debes dibujar el árbol de llamadas marcando de forma distinta aquellas soluciones que no se ramifican por no ser prometedoras.

Examen segunda convocatoria de Esquemas Algorítmicos (IG24)

6 de septiembre de 2005

1. Voraces — 2,5 puntos

Conocemos un algoritmo voraz que resuelve eficientemente el problema de la selección de actividades. Nos interesa saber si existen también soluciones voraces para la *siguiente variante* de dicho problema: disponemos de una gran sala que alquilamos para la realización de actividades y que podemos dividir mediante paneles para la realización simultánea de varios actos. La sala tiene una capacidad máxima para M personas (independientemente de que se divida o no). Por otra parte, tenemos una propuesta de todas las actividades que podemos realizar en la sala durante un día. Cada actividad ocupa un intervalo de tiempo (con una hora de inicio s y una hora de finalización t) y tiene un número máximo de participantes m.

El problema consiste en averiguar cuál es la combinación con mayor número de actividades que podemos realizar en la sala en ese día. No importa que se realicen varias actividades simultáneamente, siempre y cuando no se supere en ningún momento el aforo máximo de la sala. Asumimos que la sala se puede reconfigurar en cualquier momento de forma inmediata y que los tiempos de comienzo y finalización de las actividades coinciden con las horas en punto.

Para resolver el problema hemos diseñado las siguientes estrategias voraces:

a) Ordenar las actividades *de menor a mayor instante de finalización*. Siguiendo dicho orden, considerar cada actividad y *seleccionarla* siempre que con ello no se supere la capacidad máxima de la sala en alguna de sus horas de realización.

b) Ordenar las actividades *de menor a mayor número de participantes*. Siguiendo dicho orden, considerar cada actividad y *seleccionarla* siempre que con ello no se supere la capacidad máxima de la sala en alguna de sus horas de realización.

c) Considerar inicialmente todas las actividades como seleccionadas. A continuación, para cada hora (desde las 00h hasta las 24h) comprobar si se supera el aforo máximo de la sala. Si se supera el aforo en dicha hora, eliminar de forma repetida la actividad con más participantes de las seleccionadas hasta que quepan en la sala los participantes de todas las actividades que aún permanecen seleccionadas.

Indica y justifica, *para cada una de las estrategias propuestas*:

1) Si encuentra o no la solución óptima siempre. En este segundo caso, debes justificarlo con un contraejemplo.

2) El coste temporal de la estrategia. Indica para ello las estructuras de datos que utilizarías en la implementación de un algoritmo para la estrategia, describiendo los detalles del mismo que den lugar al coste señalado.

2. Divide y vencerás — 2,5 puntos

Sea v un vector *ordenado* de n elementos que pueden aparecer repetidos dispuestos en orden decreciente ($v[0] \geq v[1] \geq \ldots \geq v[n-1]$). La función `cuenta` calcula mediante divide y vencerás el número de veces que aparece el elemento x en v:

```
def _cuenta(v,i,k,x):
  if k-i==0: return 0
  elif k-i==1:
    if x==v[i]: return 1
    else:       return 0
  else:
    j=(k+i)/2
    if x > v[j]:    return _cuenta(v,i,j,x)
    elif x < v[j]:  return _cuenta(v,j+1,k,x)
    else:           return _cuenta(v,i,j,x)+_cuenta(v,j,k,x)
def cuenta(v,x):
  return _cuenta(v,0,len(v),x)
```

a) ¿Es correcto el algoritmo? Si crees que no, corrígelo para que, aplicando divide y vencerás, funcione adecuadamente.

b) Indica si el algoritmo presenta un comportamiento uniforme o existe un mejor y un peor caso. Si ocurre esto último, señala en qué condiciones (ante qué tipos de entradas) puede el algoritmo comportarse en el mejor de los casos y en qué condiciones en el peor y da algún ejemplo de cada una de esas entradas.

c) Realiza un análisis de complejidad temporal y espacial del algoritmo presentado, indicando los costes en el mejor y peor de los casos si los hubiera. Justifica los costes alcanzados.

d) Es posible refinar el algoritmo original introduciendo algunas modificaciones que permiten obtener una versión del mismo que presenta un menor coste temporal. Hazlo y di cuál es ese coste. (Por supuesto, el algoritmo resultante debe seguir la estrategia de divide y vencerás).

   Pistas: Una posible vía de ataque es detectar cuál es la parte del algoritmo que contribuye en mayor medida al coste e intentar que no sea así. Otra opción es estudiar aquellos casos en los que el fragmento de vector analizado presenta una solución directa al problema.

e) Propón un algoritmo iterativo que resuelva el mismo problema sin utilizar la técnica de divide y vencerás (no es necesario que lo implementes, tan sólo descríbelo). Indica el coste asintótico que presenta y si, atendiendo a dicho coste, puede considerarse mejor o peor que el anterior.

3. Programación dinámica — 3,5 puntos

El problema del *trayecto más probable en el río Congo* puede resolverse mediante un algoritmo de programación dinámica. Recordemos su planteamiento inicial: a lo largo del río Congo hay E embarcaderos (numerados del 1 al E). Es posible ir directamente en canoa desde un embarcadero a cualquiera de los dos siguientes en la dirección de la corriente, pero no más allá del segundo sin hacer escala previa. Para cualquier embarcadero $i = 1 \dots E-1$ conocemos la probabilidad de que un viajero que se encuentra en i decida ir directamente al embarcadero j, y es $p(j|i)$ (por la definición del problema, j sólo puede ser $i+1$ o $i+2$, excepto cuando $i = E-1$, en que j es $i+1$). Se nos pedía averiguar qué trayecto entre el embarcadero 1 y el E es el que sigue con más probabilidad un viajero.

Considera la siguiente variante del problema: nos interesa conocer el trayecto más probable entre el primer y el último embarcadero que *realice exactamente N escalas* (incluidos los embarcaderos origen y destino). Por tanto, desarrolla un algoritmo mediante *programación dinámica* que calcule y devuelva el trayecto más probable que atraviesa N embarcaderos con las restricciones indicadas. Para ello:

a) Presenta una formalización del problema en términos de optimización (conjunto de soluciones factibles y función objetivo),

b) plantea la ecuación recursiva que calcula el valor del trayecto más probable según las condiciones previamente descritas, indicando cuál debe ser la llamada que se efectúe a dicha función para obtener el valor óptimo, y

c) diseña un algoritmo iterativo que devuelva *el trayecto óptimo* (no sólo su valor), justificando su coste temporal y espacial.

d) ¿Podría haber instancias del problema para las que no existiera una solución factible? Razona la respuesta y, si es así, pon algún ejemplo de instancia que no tenga solución factible.

e) Otra versión del problema es aquella en la que nos interesa conocer el trayecto más probable en el que se realizan *al menos N* escalas. Indica como modificarías la solución recursiva que has planteado en el apartado b) para adaptarla a esta variante. Comenta si este hecho tiene incidencia en los costes asintóticos del algoritmo.

4. Búsqueda con retroceso — 1,5 puntos

Dados n números enteros sin signo almacenados en una lista D y un valor T, se nos pide determinar el signo (positivo o negativo) que debe tener cada uno de los n números de D para que, al sumarlos todos ellos, obtengamos el valor T.

Como solución al problema debemos devolver una lista de n componentes, cada uno de los cuales será 1 o -1 en función de si el número que esté en la correspondiente posición de D es positivo o negativo, respectivamente. En caso de no encontrar solución, deberemos devolver el valor especial `None`.

A continuación, se presenta un posible algoritmo de *búsqueda con retroceso* que resuelve este problema:

```
def signos(D,T):
  s = [None] * len(D)
  def backtracking(i):
    if es_completa(i):
      if es_factible(): return s
    else:
      for s[i] in [1, -1]:
        if es_prometedora(i):
          found = backtracking(i+1)
          if found != None: return found
    return None
 return backtracking(0)
```

Ejecutando este programa Python con los datos de entrada $D = [3, 12, 6, 3, 6, 5, 5]$ y $T = 24$, se obtiene como solución la lista $[1, 1, 1, -1, 1, 1, -1]$, que se corresponde con las operaciones: $3 + 12 + 6 + -3 + 6 + 5 + -5 = 24$. Se pide que:

a) Escribas el (pseudo)código correspondiente a la función `es_completa(i)`, es decir, que definas cuál es el criterio que permite determinar cuando un estado es una *solución completa*.

b) Escribas el (pseudo)código correspondiente a la función `es_factible()`, es decir, que definas cuál es el criterio que permite determinar si una *solución completa* es una *solución factible* o no.

c) Escribas el (pseudo)código correspondiente a la función `es_prometedora(i)`, es decir, que definas algún criterio que permita determinar si una *solución parcial* puede conducir a una solución factible (en cuyo caso se ramifica) o no.

d) A la vista del código desarrollado en los apartados anteriores, ¿cuál sería la complejidad espacial del algoritmo desarrollado? ¿Y la complejidad temporal en el *mejor de los casos*?

e) ¿Crees que es posible mejorar el coste temporal introduciendo alguna variable o estructura de datos adicional o haciendo algún tipo de preproceso? Si es así, describe con claridad la mejora o mejoras que introducirías en el algoritmo e indica como afectaría a los costes previamente calculados.

f) Realiza una traza de ejecución de este algoritmo para los datos de entrada: $D = [12, 5, 3, 2, 10]$ y $T = 6$. Debes dibujar el árbol de llamadas efectuadas, incluyendo solamente aquellos estados que se generan durante el proceso de búsqueda. Marca de forma especial los estados que se estudian pero se descartan por no superar la condición de ser prometedores.