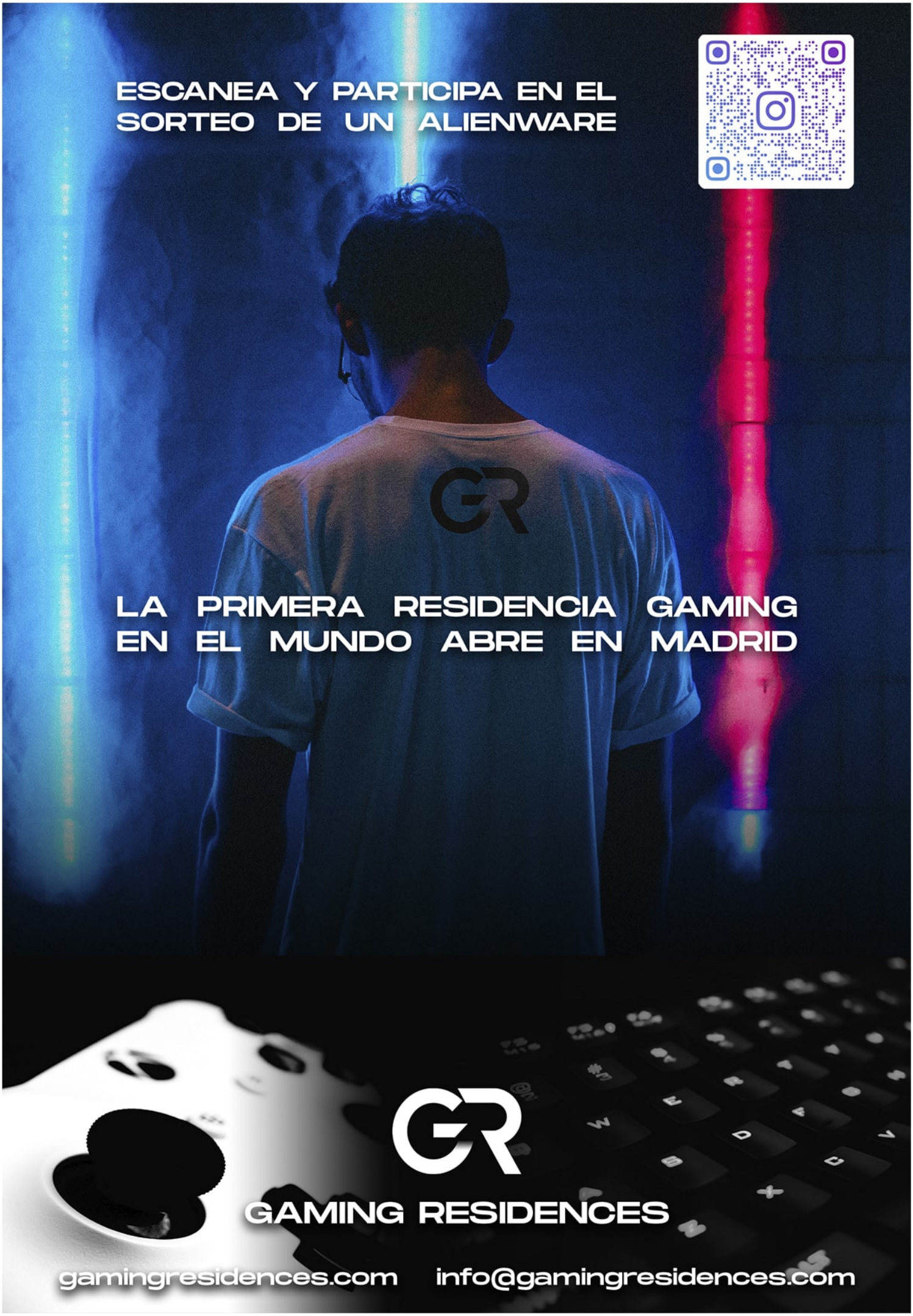
Relacion-de-Problemas-Greedy-res...

PruebaAlien

Algorítmica

2º Grado en Ingeniería Informática

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación
Universidad de Granada


ESCANEA Y PARTICIPA EN EL SORTEO DE UN ALIENWARE
LA PRIMERA RESIDENCIA GAMING EN EL MUNDO ABRE EN MADRID
GAMING RESIDENCES
gamingresidences.com
info@gamingresidences.com

Relación de ejercicios resueltos sobre el Enfoque Greedy

Notas:

1. Los ejercicios recogidos en esta relación son versiones ligeramente modificadas de algunos de los recogidos en los siguientes documentos:
   A. Sánchez: EJERCICIOS RESUELTOS. PROGRAMACIÓN III. Centro UNED-Las Rozas (Madrid) (2008-09).
   R. Guerequeta y A. Vallecillo: Técnicas de Diseño de Algoritmos. Servicio de Publicaciones de la Universidad de Málaga. Segunda Edición: Mayo 2000.
2. Estos ejercicios resueltos tienen carácter informativo y el único objetivo de ilustrar y ayudar a comprender mejor el enfoque general Greedy explicado en clase. En caso alguno son sustitutivos de las prácticas de la asignatura, que en principio, mantienen la planificación prevista.

Ejercicio 1.

Dado un conjunto de n cintas no vacías con n_i registros ordenados cada uno, se pretende mezclarlos a pares hasta lograr una única cinta ordenada. La secuencia en la que se realiza la mezcla determinará la eficiencia del proceso. Diséñese un algoritmo que busque la solución óptima minimizando el número de movimientos.

Supongamos por ejemplo 3 cintas: A con 30 registros, B con 20 y C con 10:

1. Mezclamos A con B (50 movimientos) y el resultado con C (60 movimientos), con la que realiza en total 110 movimientos.
2. Mezclamos C con B (30 movimientos) y el resultado con A (60 movimientos), con la que realiza en total 90 movimientos.

¿Hay alguna forma más eficiente de ordenar el contenido de las cintas?

Solución: El problema presenta una serie de elementos característicos de un esquema greedy:

- Por un lado, se tienen un conjunto de candidatos (las cintas) que vamos eligiendo uno a uno hasta completar determinada tarea.
- Por otro lado, el orden de elección de dichos elementos determina la corrección de la solución, de manera que para alcanzar una solución óptima es preciso seleccionar adecuadamente al candidato mediante un criterio determinado. Una vez escogido, habremos de demostrar que nos lleva a una solución óptima.

El criterio de elección de las cintas para alcanzar una solución óptima será el de elegir en cada momento aquella con menor número de registros.

Con todo, particularizando el esquema greedy general,

```
funcion voraz (C: Conjunto): conjunto
    { C es el conjunto de candidatos }
    S ← ∅          { Construimos la solución en el conjunto S }
    mientras C ≠ 0 y ¬solución (S) hacer
        x ← seleccionar (C)
        C ← C\{x};
        si factible (S ∪ {x}) entonces S ← S ∪ {x}
    si solución (S) entonces devolver S
    si no devolver "no hay solución"
```

Si

- Solución (S): El número de cintas, que es n.
- Objetivo (x): Función que devuelve la cinta con menor número de registros de entre el conjunto de cintas disponibles., y
- Factible (x) (o completable): Esta función es siempre cierta, pues cualquier orden de combinación es válido.

retocando el esquema general tenemos:

```
fun voraz (C: vector) dev (s:vector)
    para i ← 0 hasta n hacer s[i] ← 0
    i ← ∅
    mientras i ≤ n hacer
        x ← seleccionar (C)
        c[i] ← 0
        s[i] ← x
        i ← i + 1;
    fmientras
    dev S
ffun
```

Como resulta evidente se trata de un algoritmo de eficiencia cuadrática $O(n^2)$.

La demostración de la corrección, al tratarse de un ejemplo greedy muy clásico, puede encontrarse en todos los libros de Algoítmica, por ejemplo en el Horowitz-Sahni.

Ejercicio 2

Dado un tablero de ajedrez y una casilla inicial, queremos decidir si es posible que un caballo recorra todos y cada una de las casillas sin duplicar ninguna. No es necesario en este problema que el caballo vuelva a la casilla de partida. Un posible algoritmo greedy decide, en cada iteración, colocar el caballo en la casilla desde la cual domina el menor número posible de casillas aún no visitadas.

a) Implementar dicho algoritmo a partir de un tamaño de tablero $n \times n$ y una casilla inicial (x_0,y_0).

b) Buscar, utilizando el algoritmo realizado en el apartado anterior, todas las casillas iniciales para los que el algoritmo encuentra solución.

c) Basándose en los resultados del apartado anterior, encontrar el patrón general de las soluciones del recorrido del caballo.

Solución

a) Para implementar el algoritmo pedido comenzaremos definiendo las constantes y tipos que utilizaremos:

```
CONST TAMMAX = ...; (* dimension maxima del tablero *)
TYPE tablero = ARRAY[1..TAMMAX],[1..TAMMAX] OF CARDINAL;
```

Cada una de las casillas del tablero va a almacenar un número natural que indica el número de orden del movimiento del caballo en el que visita la casilla. Podrá tomar también el valor cero, indicando que la casilla no ha sido visitada aún. Inicialmente todas las casillas tomarán este valor.

Una posible implementación del algoritmo viene dada por la función *Caballo* que se muestra a continuación, la cual, dado un tablero *t*, su dimensión *n* y una posición inicial (x,y), decide si el caballo recorre todo el tablero o no.

```
PROCEDURE Caballo(VAR t:tablero; n:CARDINAL; x,y:CARDINAL):BOOLEAN;
   VAR i:CARDINAL;
BEGIN
   InicTablero(t,n); (* inicializa las casillas del tablero a 0 *)
   FOR i:=1 TO n*n DO
      t[x,y]:=i;
      IF NOT NuevoMov(t,n,x,y) AND (i<n*n-1) THEN RETURN FALSE END;
   END;
   RETURN TRUE; (* hemos recorrido las n*n casillas *)
END Caballo;
```

La función *NuevoMov* es la que va a ir calculando la nueva casilla a la que salta el caballo siguiendo la indicación del enunciado, devolviendo *FALSE* si no puede moverse:

```
PROCEDURE NuevoMov(VAR t:tablero; n:CARDINAL; VAR x,y:CARDINAL)
                                                          :BOOLEAN;
```

```
   VAR accesibles,minaccesibles:CARDINAL;
       i,solx,soly,nuevax,nuevay:CARDINAL;
BEGIN
   minaccesibles:=9;
   solx:=x; soly:=y;
   FOR i:=1 TO 8 DO
      IF Salto(t,n,i,x,y,nuevax,nuevay) THEN
         accesibles:=Cuenta(t,n,nuevax,nuevay);
         IF (accesibles>0) AND (accesibles<minaccesibles) THEN
            minaccesibles:=accesibles;
            solx:=nuevax; soly:=nuevay;
         END
      END
   END;
   x:=solx; y:=soly;
   RETURN (minaccesibles<9);
END NuevoMov;
```

Para su implementación necesitamos dos funciones auxiliares: *Salto* y *Cuenta*. La primera calcula las coordenadas de la casilla a donde salta el caballo (tiene 8 posibilidades), y devuelve si es posible realizar ese movimiento o no (puede estar ocupada o bien salirse del tablero):

```
PROCEDURE Salto(VAR t:tablero;n:CARDINAL;i:CARDINAL;
                x,y:CARDINAL;VAR nx,ny:CARDINAL) :BOOLEAN;
   (* i indica el numero del movimiento, (x,y) es la casilla
      actual, y (nx,ny) es la casilla a donde salta. *)
BEGIN
   CASE i OF
      |1: nx:=x-2; ny:=y+1;  |2: nx:=x-1; ny:=y+2;
      |3: nx:=x+1; ny:=y+2;  |4: nx:=x+2; ny:=y+1;
      |5: nx:=x+2; ny:=y-1;  |6: nx:=x+1; ny:=y-2;
      |7: nx:=x-1; ny:=y-2;  |8: nx:=x-2; ny:=y-1;
   END;
   RETURN((1<=nx)AND(nx<=n)AND(1<=ny)AND(ny<=n)AND(t[nx,ny]=0));
END Salto;
```

Dicha función intenta los movimientos en el orden que muestra la siguiente figura:

	2		3	
1				4
		X		

8				5
	7		6	

La otra función es *Cuenta*, que devuelve el número de casillas a las que el caballo puede saltar desde una posición dada:

```
PROCEDURE Cuenta(VAR t:tablero;n:CARDINAL;x,y:CARDINAL):CARDINAL;
   VAR acc,i,nx,ny:CARDINAL;
BEGIN
   acc:=0;
   FOR i:=1 TO 8 DO
      IF Salto(t,n,i,x,y,nx,ny) THEN INC(acc) END
   END;
   RETURN acc;
END Cuenta;
```

Obsérvese que hemos utilizado el paso del tablero por referencia (mediante *VAR*) en vez de por valor en todos los procedimientos aunque no se modifique el vector, para evitar su copia en la pila.

b) Para resolver esta cuestión necesitamos un programa que nos permita ir recorriendo todas las posibilidades e imprimiendo aquellas casillas iniciales desde donde se consigue solución:

```
MODULE Caballos;
   ....
   VAR t:tablero; n,i,j:CARDINAL;
BEGIN
   FOR n:=4 TO TAMMAX DO
      WrStr('Dimension = '); WrCard(n,0); WrLn();
      FOR i:=1 TO n DO FOR j:=1 TO n DO
            IF Caballo(t,n,i,j) THEN
               WrStr(' Desde: '); WrCard(i,0); WrStr(',');
               WrCard(j,0); WrStr(' tiene solucion.'); WrLn();
            END
      END END;
      WrLn();
   END
END Caballos.
```

c) La salida del programa anterior nos permite inferir un patrón general para las soluciones del problema:

- Para $n = 4$, el problema no tiene solución.
- Para $n > 4$, n par, el problema tiene solución para cualquier casilla inicial.

Ejercicio 3

Dada una secuencia de palabras p_1, p_2, ..., p_n de longitudes l_1, l_2, ..., l_n se desea agruparlas en líneas de longitud L. Las palabras están separadas por espacios cuya amplitud ideal (en milímetros) es b, pero los espacios pueden reducirse o ampliarse si es necesario (aunque sin solapamiento de palabras), de tal forma que una línea p_i, p_{i+1}, ..., p_j tenga exactamente longitud L. Sin embargo, existe una penalización por reducción o ampliación en el número total de espacios que aparecen o desaparecen. El *costo* de fijar la línea p_i, p_{i+1}, ..., p_j es $(j - i)|b' - b|$, siendo b' el ancho real de los espacios, es decir $(L - l_i - l_{i+1} - ... - l_j)/(j - i)$. No obstante, si $j = n$ (la última palabra) el costo será cero a menos que $b' < b$ (ya que no es necesario ampliar la última línea).

En primer lugar, necesitamos plantear un algoritmo greedy para resolver el problema, implementarlo y dar un ejemplo donde este algoritmo no encuentre solución óptima o bien demostrar que tal ejemplo no existe.

Por otra lado, consideraremos el caso especial de usar una impresora de líneas, en donde por sus características especiales el valor óptimo de b es 1 y no se puede producir reducción de espacios (ya que b no puede ser 0).

Solución

Para resolver este problema mediante un algoritmo greedy pensemos en lo que haríamos en la práctica para solucionarlo. En primer lugar, iríamos construyendo la línea empezando por la primera palabra y añadiendo las demás en orden, separándolas con espacios de tamaño óptimo b, hasta llegar a una palabra p_a (a>1) que no quepa en la línea, es decir:

$$l_1 + l_2 + ... + l_a + (a-1)*b > L.$$

Si ocurriera que $l_1 + l_2 + ... + l_a + (a-1)*b = L$, esto es, que la palabra encajara perfectamente en la línea, sencillamente imprimiríamos la línea y continuaríamos con la siguiente. Pero si no, necesitaríamos tomar una decisión: o se comprimen las palabras $p_1,...,p_{a-1}$ (recortando el tamaño de los espacios que las separan) para que

pueda caber también p_a en la línea; o bien se pasa la palabra p_a a la siguiente línea y se imprime la línea en curso, aumentando antes los espacios entre las palabras $p_1,...,p_{a-1}$ para que la línea tenga exactamente longitud L.

El algoritmo greedy simplemente va a escoger aquella opción que suponga un menor coste. Obsérvese que estamos ante un típico algoritmo greedy, pues siempre toma su decisión basado en una optimización local y nunca "guarda historia".

Para implementar tal algoritmo, vamos a disponer de un vector que almacena las longitudes de las palabras, y la solución vamos a darla como un vector de registros, uno por cada línea. Cada registro contiene los índices (número de orden) de las palabras que comienzan y terminan la línea, el tamaño de los espacios entre las palabras y el coste de la línea. Esto da lugar al siguiente algoritmo:

```
CONST MAXPALABRAS = ...;
      MAXLINEAS   = MAXPALABRAS; (* para cubrir el peor caso *)
TYPE  REGISTRO=  RECORD
                   primera,ultima:CARDINAL;
                   espacio,coste:REAL;
                 END;
      SOLUCION=  ARRAY [1..MAXLINEAS] OF REGISTRO;
      LONGPALS=  ARRAY [1..MAXPALABRAS] OF CARDINAL;

PROCEDURE Parrafo(L:CARDINAL;n:CARDINAL;b:CARDINAL;VAR l:LONGPALS;
                  VAR sol:SOLUCION):CARDINAL;
    (* L es la longitud de la linea, n el numero de palabras, b el
       tamano optimo de los espacios, l es el vector con las
       longitudes de las n palabras, y en sol almacena la solucion.
       Devuelve el numero de lineas que ha necesitado *)

  VAR  tamanopalabras:CARDINAL; (* long de palabras de la linea *)
       tamanolinea:CARDINAL; (* tamano de la linea en curso *)
       nlinea:CARDINAL;      (* linea en curso *)
       npalabra:CARDINAL;    (* palabra en curso *)
       nespacios:CARDINAL;   (* num. espacios linea en curso *)

  PROCEDURE Espacio(L,tampalabras,nesp:CARDINAL):REAL;
  (* devuelve cero si nesp = 0, o bien un numero mayor que 1 *)
  BEGIN
     IF nesp=0 THEN RETURN 0.0 END;
     RETURN REAL(L-tampalabras)/REAL(nesp);
  END Espacio;

  PROCEDURE ResetContadores(linea,npal:CARDINAL);
  BEGIN
     IF npal<=n THEN (* para la ultima palabra no hacemos nada *)
        sol[linea].primera:=npal;
        sol[linea].coste:=0.0;
```

```
      tamanopalabras:=l[npal];
      tamanolinea:=l[npal];
      nespacios:=0
    END;
  END ResetContadores;

  PROCEDURE Coste(L,b,tampalabras,nesp:CARDINAL):REAL;
    VAR bprima:REAL;
  BEGIN
    bprima:=Espacio(L,tampalabras,nesp);
    IF bprima>REAL(b) THEN RETURN REAL(nesp)*(bprima-REAL(b))
    ELSE RETURN REAL(nesp)*(REAL(b)-bprima)
    END;
  END Coste;

  PROCEDURE CerrarLinea(linea,npal:CARDINAL);
  BEGIN
    sol[linea].ultima:=npal-1;
    sol[linea].espacio:=Espacio(L,tamanopalabras,nespacios);
    sol[linea].coste:=Coste(L,b,tamanopalabras,nespacios);
  END CerrarLinea;
BEGIN      (* programa principal del procedimiento Parrafo *)
  nlinea:=1;
  ResetContadores(nlinea,1); (* metemos la primera palabra *)
  npalabra:=2;
  WHILE (npalabra<=n) DO
    IF tamanolinea+b+l[npalabra]<=L THEN (* cabe *)
      INC(tamanolinea,b+l[npalabra]);
      INC(tamanopalabras,l[npalabra]);
      INC(nespacios)
    ELSE  (* no cabe de forma optima *)
      IF (tamanopalabras+l[npalabra]+nespacios+1)>L THEN
        (* no cabe en cualquier caso: la pasamos a otra linea *)
        CerrarLinea(nlinea,npalabra);
        INC(nlinea);       (* reinicializamos contadores *)
        ResetContadores(nlinea,npalabra);
      ELSE (* podria caber. Tenemos que tomar una decision *)
        IF Coste(L,b,tamanopalabras,nespacios)>=
        Coste(L,b,tamanopalabras+l[npalabra],nespacios+1) THEN
          INC(npalabra);  (* la metemos en la linea en curso *)
        END;
        (* si no, la pasamos a la otra linea *)
        CerrarLinea(nlinea,npalabra);
        INC(nlinea);
        ResetContadores(nlinea,npalabra);
      END
```

```
      END;
      INC(npalabra);
    END;
    IF sol[nlinea].primera=0 THEN RETURN nlinea-1 END;
    IF sol[nlinea].ultima=0 THEN CerrarLinea(nlinea,npalabra) END;
    RETURN nlinea;
  END Parrafo;
```

La eficiencia de este algoritmo es de orden O(n), debido al bucle que se repite a lo más n–1 veces (una por cada palabra menos la primera y aquellas que decidamos meter comprimiendo la línea), y que dentro del bucle todas las operaciones que se realizan son de complejidad constante.

En cuanto a su funcionamiento, desafortunadamente no podemos afirmar que encuentre solución óptima en todos los casos, como pone de manifiesto el siguiente ejemplo.

Supongamos que L = 26, b = 2, y que disponemos de n = 7 palabras, cuyas longitudes son 10, 10, 4, 8, 10, 12 y 12.

El algoritmo anterior, tras meter las dos primeras palabras en la primera línea, tiene que tomar una decisión en cuanto a si la tercera palabra (de longitud 4) debe estar en la primera línea o no. En caso de estar, hay que comprimir los espacios, lo que ocasiona un coste de valor 2; por otro lado, si la pasa a la segunda línea es necesario expandir el espacio entre las dos palabras, lo que supone un coste de valor 4.

Ante esta disyuntiva, el algoritmo decide incluirla en la primera línea, lo que da lugar a la siguiente descomposición en líneas (expresadas con paréntesis):

(10, 10, 4), (8, 10), (12, 12).

El coste global de esta descomposición es 8 (=2+6+0), mientras que si hubiera tomado la alternativa que inicialmente tenía más coste hubiera llegado a la descomposición:

(10, 10), (4, 8, 10), (12, 12)

cuyo coste global es 4 (= 4+0+0).

El motivo del fallo de este algoritmo es su "glotonería", como le ocurre a todos los algoritmos greedy. En general este problema lo va a tener cualquier algoritmo que, sin disponer de posibilidades de decidir el orden en el que se van produciendo las entradas, no sea capaz de hacer sacrificios locales para obtener resultados globales óptimos.

Ejercicio 4

Necesitamos en este problema calcular la matriz producto M de n matrices dadas $M=M_1M_2...M_n$. Por ser asociativa la multiplicación de matrices, existen muchas formas posibles de realizar esa operación, cada una con un coste asociado (en términos del número de multiplicaciones escalares). Si cada M_i es de dimensión d_{i-1}xd_i ($1 \le i \le n$), multiplicar M_iM_{i+1} requiere $d_{i-1}d_id_{i+1}$ operaciones.

El problema consiste en encontrar el mínimo número de operaciones necesario para calcular el producto M.

En general, el coste asociado a las distintas formas de multiplicar las n matrices puede ser bastante diferente de unas a otras. Por ejemplo, para $n = 4$ y para las matrices M_1, M_2, M_3 y M_4 cuyos órdenes son:

$$M_1(30\times1),\ M_2(1\times40),\ M_3(40\times10),\ M_4(10\times25)$$

hay cinco formas distintas de multiplicarlas, y sus costes asociados (en términos de las multiplicaciones escalares que necesitan) son:

$((M_1M_2)M_3)M_4$	$= 30\cdot1\cdot40 + 30\cdot40\cdot10 + 30\cdot10\cdot25$	$= 20.700$
$M_1(M_2(M_3M_4))$	$= 40\cdot10\cdot25 + 1\cdot40\cdot25 + 30\cdot1\cdot25$	$= 11.750$
$(M_1M_2)(M_3M_4)$	$= 30\cdot1\cdot40 + 40\cdot10\cdot25 + 30\cdot40\cdot25$	$= 41.200$
$M_1((M_2M_3)M_4)$	$= 1\cdot40\cdot10 + 1\cdot10\cdot25 + 30\cdot1\cdot25$	$= 1.400$
$(M_1(M_2M_3))M_4$	$= 1\cdot40\cdot10 + 30\cdot1\cdot10 + 30\cdot10\cdot25$	$= 8.200$

Como puede observarse, la mejor forma necesita casi treinta veces menos multiplicaciones que la peor, por lo cual es importante elegir una buena asociación. Podríamos pensar en calcular el coste de cada una de las opciones posibles y escoger la mejor de entre ellas antes de multiplicar. Sin embargo, para valores grandes de n esta estrategia es inútil, pues el número de opciones crece exponencialmente con n. De hecho, el número de opciones posible sigue la sucesión de los números de Catalan:

$$T(n) = \sum_{i=1}^{n-1} T(i)T(n-i) = \frac{1}{n}\binom{2n-2}{n-1} \in \Theta\left(\frac{4^n}{\sqrt{n}}\right).$$

Parece entonces muy útil la búsqueda de un algoritmo greedy que resuelva nuestro problema. La siguiente lista presenta cuatro estrategias diferentes:

a) Multiplicar primero las matrices M_iM_{i+1} cuya dimensión común d_i sea la menor entre todas, y repetir el proceso.

b) Multiplicar primero las matrices M_iM_{i+1} cuya dimensión común d_i sea la mayor entre todas, y repetir el proceso.

c) Realizar primero la multiplicación de las matrices M_iM_{i+1} que requiera menor número de operaciones ($d_{i-1}d_id_{i+1}$), y repetir el proceso.

d) Realizar primero la multiplicación de las matrices M_iM_{i+1} que requiera mayor número de operaciones ($d_{i-1}d_id_{i+1}$), y repetir el proceso.

Queremos determinar si alguna de las estrategias propuestas encuentra siempre solución óptima. Como es habitual en los algoritmos greedy para comprobar su funcionamiento, sería necesario una demostración formal o bien dar un contraejemplo que justifique la respuesta.

Solución (☺)

Lamentablemente, ninguna de las estrategias presentadas encuentra la solución óptima. Por tanto, para todas es posible dar un contraejemplo en donde el algoritmo falla.

a) La primera estrategia consiste en multiplicar siempre primero las matrices $M_i\, M_{i+1}$ cuya dimensión común d_i es la menor entre todas. Pero observando el ejemplo anterior esta estrategia se muestra errónea, pues corresponde al producto $(M_1M_2)(M_3M_4)$, que resulta ser el peor de todos.

b) Si multiplicamos siempre primero las matrices M_iM_{i+1} cuya dimensión común d_i es la mayor entre todas encontramos la solución óptima para el ejemplo anterior, pero existen otros ejemplos donde falla esta estrategia, como el siguiente:

   Sean $M_1(2\times5)$, $M_2(5\times4)$ y $M_3(4\times1)$. Según esta estrategia, el producto escogido como mejor sería $(M_1M_2)M_3$, con un coste de 48 $(2\cdot5\cdot4 + 2\cdot4\cdot1)$. Sin embargo, el producto $M_1(M_2M_3)$ tiene un coste asociado de 30 $(5\cdot4\cdot1 + 2\cdot5\cdot1)$, menor que el anterior.

c) Si decidimos realizar siempre primero la multiplicación de las matrices $M_i\, M_{i+1}$ que requiera menor número de operaciones, encontraríamos la solución óptima para los dos ejemplos anteriores, pero no para el siguiente:

   Sean $M_1(3\times1)$, $M_2(1\times100)$ y $M_3(100\times5)$. Según esta estrategia, el producto escogido como mejor sería $(M_1M_2)M_3$, de coste 1.800 $(3\cdot1\cdot100 + 3\cdot100\cdot5)$. Sin embargo, el coste del producto $M_1(M_2M_3)$ es de 515 $(1\cdot100\cdot5 + 3\cdot1\cdot5)$, menor que el anterior.

d) Análogamente, realizando siempre primero la multiplicación de las matrices $M_i\, M_{i+1}$ que requiera mayor número de operaciones vamos a encontrar la solución óptima para este último ejemplo, pero no para los dos primeros.

Ejercicio 5. Problema de la asignación de tareas

Supongamos que disponemos de n trabajadores y n tareas. Sea $b_{ij} > 0$ el coste de asignarle el trabajo j al trabajador i. Una asignación de tareas puede ser expresada como una asignación de los valores 0 ó 1 a las variables x_{ij}, donde $x_{ij} = 0$ significa que al trabajador i no le han asignado la tarea j, y $x_{ij} = 1$ indica que sí. Una asignación válida es aquella en la que a cada trabajador sólo le corresponde una tarea y cada tarea está asignada a un trabajador. Dada una asignación válida, definimos el *coste* de dicha asignación como:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} x_{ij} b_{ij} .$$

Diremos que una asignación es óptima si es de mínimo coste. Cara a diseñar un algoritmo greedy para resolver este problema podemos pensar en dos estrategias distintas: asignar cada trabajador la mejor tarea posible, o bien asignar cada tarea al mejor trabajador disponible. Sin embargo, ninguna de las dos estrategias tiene por qué encontrar siempre soluciones óptimas. ¿Es alguna mejor que la otra?

Solución

Este es un problema que aparece con mucha frecuencia, en donde los costes son o bien tarifas (que los trabajadores cobran por cada tarea) o bien tiempos (que tardan en realizarlas). Para implementar ambos algoritmos vamos a definir la matriz de costes (b_{ij}):

```
TYPE COSTES = ARRAY[1..n],[1..n] OF CARDINAL;
```

que forma parte de los datos de entrada del problema, y la matriz de asignaciones (x_{ij}), que es la que buscamos:

```
TYPE ASIGNACION = ARRAY[1..n],[1..n] OF BOOLEAN;
```

Con esto, el primer algoritmo puede ser implementado como sigue:

```
PROCEDURE AsignacionOptima(VAR b:COSTES; VAR x:ASIGNACION);
   VAR trabajador,tarea:CARDINAL;
```

```
BEGIN
   FOR trabajador:=1 TO n DO (* inicializamos la matriz solucion *)
      FOR tarea:=1 TO n DO
         x[trabajador,tarea]:=FALSE
      END
   END;
   FOR trabajador:=1 TO n DO
      x[trabajador,MejorTarea(b,x,trabajador)]:=TRUE
   END
END AsignacionOptima;
```

La función *MejorTarea* es la que busca la mejor tarea aún no asignada para ese trabajador:

```
PROCEDURE MejorTarea (VAR b:COSTES; VAR x:ASIGNACION;
                          i:CARDINAL):CARDINAL;
   VAR tarea,min,mejortarea:CARDINAL;
BEGIN
   min:=MAX(CARDINAL);
   FOR tarea:=1 TO n DO
      IF (NOT YaEscogida(x,i,tarea))AND(b[i,tarea]<min) THEN
         min:=b[i,tarea];
         mejortarea:=tarea
      END
   END;
   RETURN mejortarea;
END MejorTarea;
```

Por último, la función *YaEscogida* decide si una tarea ya ha sido asignada previamente:

```
PROCEDURE YaEscogida(VAR x:ASIGNACION;
                     trabajador,tarea:CARDINAL):BOOLEAN;
   VAR i:CARDINAL;
BEGIN
   FOR i:=1 TO trabajador-1 DO
      IF x[i,tarea] THEN RETURN TRUE END
   END;
   RETURN FALSE;
END YaEscogida;
```

Lamentablemente, este algoritmo ávido no funciona para todos los casos como pone de manifiesto la siguiente matriz de valores:

		Tarea		
		1	2	3
	1	16	20	18
Trabajador	2	11	15	17
	3	17	1	20

Para ella, el algoritmo produce una matriz de asignaciones en donde los "unos" están en las posiciones (1,1), (2,2) y (3,3), esto es, asigna la tarea *i* al trabajador *i* (*i* = 1, 2 ,3), con un valor de la asignación de 51 (= 16 + 15 + 20). Sin embargo la asignación óptima se consigue con los "unos" en posiciones (1,3), (2,1) y (3,2), esto es, asigna la tarea 3 al trabajador 1, la 1 al trabajador 2 y la tarea 2 al trabajador 3, con un valor de la asignación de 30 (= 18 + 11 + 1).

Si utilizamos la segunda estrategia nos encontramos en una situación análoga. En primer lugar, su implementación es:

```
PROCEDURE AsignacionOptima2(VAR b:COSTES; VAR x:ASIGNACION);
   VAR trabajador,tarea:CARDINAL;
BEGIN
   FOR trabajador:=1 TO n DO (* inicializamos la matriz solucion *)
      FOR tarea:=1 TO n DO
         x[trabajador,tarea]:=FALSE
      END
   END;
   FOR tarea:=1 TO n DO
      x[MejorTrabajador(b,x,tarea),tarea]:=TRUE
   END;
END AsignacionOptima2;
```

La función *MejorTrabajador* es la que busca el mejor trabajador aún no asignado para esa tarea:

```
PROCEDURE MejorTrabajador (VAR b:COSTES; VAR x:ASIGNACION;
                           i:CARDINAL):CARDINAL;
   VAR trabajador,min,mejortrabajador:CARDINAL;
BEGIN
   min:=MAX(CARDINAL);
   FOR trabajador:=1 TO n DO
     IF(NOT YaEscogido(x,i,trabajador))AND(b[trabajador,i]<min)THEN
        min:=b[trabajador,i];
        mejortrabajador:=trabajador
      END
   END;
   RETURN mejortrabajador;
END MejorTrabajador;
```

Por último, la función *YaEscogido* decide si un trabajador ya ha sido asignado previamente:

```
PROCEDURE YaEscogido(VAR x:ASIGNACION;
                     trabajador,tarea:CARDINAL):BOOLEAN;
  VAR i:CARDINAL;
BEGIN
  FOR i:=1 TO tarea-1 DO
    IF x[trabajador,i] THEN RETURN TRUE END
  END;
  RETURN FALSE;
END YaEscogido;
```

Lamentablemente, este algoritmo greedy tampoco funciona para todos los casos como pone de manifiesto la siguiente matriz de valores:

		Tarea		
		1	2	3
Trabajador	1	16	11	17
	2	20	15	1
	3	18	17	20

Para ella, el algoritmo produce una matriz de asignaciones en donde los "unos" vuelven a estar en las posiciones (1,1), (2,2) y (3,3), con un valor de la asignación de 51 (=16+15+20). Sin embargo la asignación óptima se consigue con los "unos" en posiciones (3,1), (1,2) y (2,3), con un valor de la asignación de 30 (=18+11+1).

Respecto a la pregunta de si una estrategia es mejor que la otra, la respuesta es que no. La razón es que las soluciones son simétricas. Aún más, una es la imagen especular de la otra. Por tanto, si suponemos equiprobables los valores de las matrices, ambos algoritmos van a tener el mismo número de casos favorables y desfavorables.

Ejercicio 6. Camino al sur

Un camionero conduce desde Bilbao a Granada 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 pararse a repostar el menor número de veces posible.

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

Solución

Supondremos que existen G gasolineras en la ruta que sigue el camionero entre Bilbao y Granada, incluyendo una en la ciudad destino, y que están numeradas del 0 (gasolinera en Bilbao) a G–1 (la situada en Granada).

Supondremos además que disponemos de un vector con la información que tiene el camionero sobre las distancias entre ellas:

```
TYPE DISTANCIA = ARRAY [1..G-1] OF CARDINAL;
```

de forma que el i-ésimo elemento del vector indica los kilómetros que hay entre las gasolineras i–1 e i. Para que el problema tenga solución hemos de suponer que ningún valor de ese vector es mayor que el número n de kilómetros que el camión puede recorrer sin repostar.

Con todo esto, el algoritmo greedy pedido va a consistir en intentar recorrer el mayor número posible de kilómetros sin repostar, esto es, tratar de ir desde cada gasolinera en donde se pare a repostar a la más lejana posible, así hasta llegar al destino.

Para demostrar la validez de este algoritmo greedy, sean $x_1,x_2,...,x_s$ las gasolineras en donde este algoritmo decide que hay que parar a repostar, y sea $y_1,y_2,...,y_t$ otro posible conjunto solución de gasolineras. Llamaremos X a un camión que sigue la primera solución, e Y a un camión que se guía por la segunda. Sea N el número total de kilómetros a recorrer (distancia entre las dos ciudades), y sea $D[i]$ la distancia recorrida por el camionero hasta la i-ésima gasolinera ($1 \le i \le G$–1). Es decir,

$$D[i] = \sum_{k=1}^{i} d[k] \quad \text{y} \quad D[G-1] = N.$$

Lo que tenemos que demostrar es que $s \le t$, puesto que lo que queríamos minimizar era el número de paradas a realizar. Para probarlo, basta con demostrar que $x_k \ge y_k$ para todo k.

En primer lugar, como ambas descomposiciones son distintas, sea k el primer índice tal que $x_k \neq y_k$. Podemos suponer sin perder generalidad que $k = 1$, puesto que hasta x_{k-1} los viajes son iguales, y en la gasolinera x_{k-1} ambos camiones rellenan su tanque completamente.

Por la forma en que funciona el algoritmo, si $x_1 \neq y_1$ entonces $x_1 > y_1$, pues x_1 era la gasolinera más alejada a donde podía viajar el camionero sin repostar.

Además, también se tiene que $x_2 \geq y_2$, pues x_2 era la gasolinera más alejada a donde podía viajar desde x_1 el camionero sin repostar. Para probar este hecho, supongamos por reducción al absurdo que y_2 fuera estrictamente mayor que x_2. Pero si Y consigue ir desde y_1 a y_2 es que hay menos de n kilómetros entre ellas, es decir,

$$D[y_2] - D[y_1] < n.$$

Por tanto desde x_1 también hay menos de n kilómetros hasta y_2, esto es,

$$D[y_2] - D[x_1] < n$$

puesto que $D[y_1] < D[x_1]$. Entonces el método no hubiera escogido x_2 como siguiente gasolinera a x_1 sino y_2, porque el algoritmo busca siempre la gasolinera más alejada de entre las que alcanza.

Repitiendo el proceso, vamos obteniendo en cada paso que $x_k \geq y_k$ para todo k, hasta llegar a la ciudad destino, lo que demuestra la hipótesis.

El siguiente procedimiento implementa este algoritmo, devolviendo un vector que indica en qué gasolineras ha de pararse y en cuáles no:

```
TYPE SOLUCION = ARRAY [1..G-1] OF BOOLEAN;
PROCEDURE Deprisa(n:CARDINAL; VAR d:DISTANCIA; VAR sol:SOLUCION);
   VAR i,numkilometros:CARDINAL;
BEGIN
   FOR i:=1 TO G-1 DO sol[i]:=FALSE END;
   i:=0;
   numkilometros:=0;
   REPEAT
      REPEAT
         INC(i);
         numkilometros:=numkilometros+d[i];
      UNTIL (numkilometros>n) OR (i=G-1);
      IF numkilometros>n THEN  (* si nos hemos pasado... *)
         DEC(i);               (* volvemos atras una gasolinera *)
         sol[i]:=TRUE;         (* y repostamos en ella. *)
         numkilometros:=0;     (* reset contador *)
      END
   UNTIL (i=G-1);
END Deprisa;
```