El problema del barco de mercancías

Se tiene un barco de mercancías cuya capacidad de carga es de K toneladas y un conjunto de contenedores $c_1, c_2, \ldots c_n$ cuyos pesos respectivos (en toneladas) son: $p_1, p_2, \ldots p_n$. La capacidad del barco es inferior a la suma total de los pesos de los contenedores

Diseñar un algoritmo greedy que maximice el número de contenedores cargados

Lista de candidatos C: conjunto de contenedores

Lista de seleccionados S: solución parcial con los contenedores seleccionados hasta el momento.

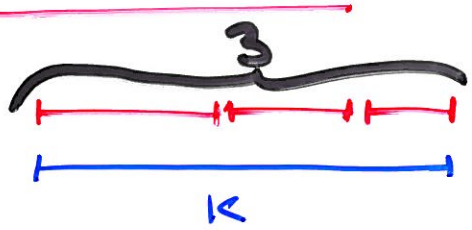
Función solución: cuando no queden elementos / candidatos factibles

Función objetivo: nº elementos en S

Función selección: Posibilidades:

1. Coger el siguiente contenedor de menor peso
2. Coger el siguiente contenedor de mayor peso

Selección 2



Cabrían mas!!

Optamos por la selección 1

Algoritmo

```
MaxMont (C, S)
{
    S = {∅}
    P = ∅
    Mientras (C ≠ ∅) hacer    n veces
    {
        x = buscarmínimo(C)    // n, n-1, n-2
        C = C - {x}
        Si (p + peso(x) ≤ k)
        {
            S = S ∪ {x}
            p = p + peso(x)
        }
    }
}
```

O(1)

$O(n^2)$

pero podría ser nlogn si se hace una ordenación previa de los contenedores por peso e ir seleccionandolos mientras quepan

¿ Y si hubiera que diseñar un algoritmo greedy para maximizar el número de toneladas cargado?

Encontrar contraejemplos que prueben que en este caso el enfoque greedy no es adecuado

El problema del fontanero

Un fontanero necesita hacer n reparaciones urgentes, y sabe de antemano el tiempo que le va a llevar cada una de ellas: en la tarea i-ésima tardará t_i minutos.

Se trata de decidir el orden en el que atenderá los avisos para minimizar el tiempo medio de espera de los clientes

Si E_i es el tiempo de espera del cliente i-ésimo hasta ver reparada su avería por completo, se trataría de minimizar la expresión $E(n) = \sum_{i=1}^{n} E_i$

El fontanero siempre tardará el mismo tiempo global $T = t_1 + t_2 + \cdots + t_n$ en realizar todas las reparaciones independientemente de cómo las ordene. Sin embargo, los tiempos de espera de los clientes sí dependen de esta ordenación

Si se mantuviese la ordenación original de las tareas $(1, 2, \ldots, n)$ la expresión de los tiempos de espera de los clientes viene dada por:

$$E_1 = t_1$$
$$E_2 = t_1 + t_2$$
$$\vdots$$
$$E_n = t_1 + t_2 + \ldots + t_n$$

Tenemos que encontrar una permutación de las tareas donde se minimice $E(n)$, es decir minimizar:

$$E(n) = \sum_{i=1}^{n} E_i =$$

Solución greedy

La permutación óptima es aquella en la que los avisos se atienden en orden creciente de sus tiempos de reparación

$$\sum_{i=1}^{n} E_i = t_1 + (t_1 + t_2) + (t_1 + t_2 + t_3) + \ldots + (t_1 + t_2 + \ldots + t_n) = n t_1 + (n-1) t_2 + \ldots + \cdot = \sum_{k=1}^{n} (n-k-1)\, t_k$$

PRODUCTO DE MATRICES

$M_1 \rightarrow 10 \times 20$

$M_2 \rightarrow 20 \times 50$

$M_3 \rightarrow 50 \times 1$

$M_4 \rightarrow 1 \times 100$

$M_1 \times M_2 \times M_3 \times M_4$

a)

$M_1 \times M_2 \times M_3 \times M_4$

$(10 \times 20) \quad (20 \times 50) \quad (50 \times 1) \quad (1 \times 100)$

M_{12} [10.000] (10×50)

M_{123} [500] (10×1)

M_{1234} [1.000] (10×100)

[11.500]

b)

$M_1 \times M_2 \times M_3 \times M_4$

$(10 \times 20) \quad (20 \times 50) \quad (50 \times 1) \quad (1 \times 100)$

M_{23} [1.000] (20×1)

M_{123} [200] (10×1)

M_{1234} [1.000] (10×100)

[2.200]

Multiplicación óptima de matrices

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

$((M_1M_2)M_3)M_4 = 30\cdot1\cdot40 + 30\cdot40\cdot10 + 30\cdot10\cdot25 = 20.700$

$(M_1(M_2(M_3M_4)) = 40\cdot10\cdot25 + 1\cdot40\cdot25 + 30\cdot1\cdot25 = 11.750$

$(M_1M_2)(M_3M_4) = 30\cdot1\cdot40 + 40\cdot10\cdot25 + 30\cdot40\cdot25 = 41.200$

$M_1((M_2M_3)M_4) = 1\cdot40\cdot10 + 1\cdot10\cdot25 + 30\cdot1\cdot25 = 1.400$

$(M_1(M_2M_3))M_4 = 1\cdot40\cdot10 + 30\cdot1\cdot10 + 30\cdot10\cdot25 = 8.200$

Posibles estrategias

1. Multiplicar primero las matrices M_i M_{i+1} cuya dimensión común d_i sea la menor entre todas y repetir el proceso

$(M_1M_2)(M_3M_4)$ Es peor resultado

2. Igual que 1 cambiando menor por mayor

$M_1((M_2M_3)M_4)$ optimo en el ej. pas.

Pero: $M_1[2\times5] * M_2[5\times4] * M_3[4\times1]$

Haría $(M_1M_2)M_3$ $(2\cdot5\cdot4 + 2\cdot4\cdot1 = 48)$ pero el producto $M_1(M_2M_3)$ $(5\cdot4\cdot1 + 2\cdot5\cdot1 = 30)$ es menor

3. Realizar primero la multiplicación de las matrices $M_i M_{i+1}$ que requiera menor número de operaciones $[d_{i-1}\ d_i\ d_{i+1}]$ y repetir el proceso

$M_1((M_2 M_3) M_4)$
$M_1(M_2 M_3)$
} óptimo en los 2 ejs.

pero:

$M_1[3\times1] * M_2[1\times100] * M_3[100\times5]$

haría: $(M_1 M_2) M_3$ $(3\cdot1\cdot100 + 3\cdot100\cdot5 = 1800)$

pero: $M_1(M_2 M_3)$ $(1\cdot100\cdot5 + 3\cdot1\cdot5 = 515)$ es mejor

4. Realizar primero la multiplicación de las matrices $M_i M_{i+1}$ que requiera mayor número de operaciones $[d_{i-1}\ d_i\ d_{i+1}]$ y repetir el proceso

óptimo en el último ejemplo pero no en los 2 anteriores

Estrategia greedy usual: la 3

Problema de ejecución de tareas

Tenemos que ejecutar un conjunto de n tareas, cada una de las cuales requiere un tiempo unitario. En un instante T = 1, 2, .. podemos ejecutar únicamente una tarea. La tarea (i) produce un beneficio (g_i) (>0) solo en el caso de que sea ejecutada en un instante anterior o igual a d_i.

Utilizando la técnica greedy, encontrar un algoritmo que nos permita seleccionar el conjunto de tareas a realizar de forma que nos aseguremos que tenemos la mayor ganancia posible.

Aplicarlo al siguiente ejemplo:

i	1	2	3	4	
g_i	50	10	15	30	
d_i	2	1	2	1	

* Lista de candidatos C: conjunto de tareas
* Lista de seleccionados S: Las tareas seleccionadas hasta el momento
* Función solución: Cuando no queden más tareas factibles
* Función objetivo: Beneficios acumulados por las tareas en S: $B = \sum_{i \in S} g_i$
* Función de factibilidad: $(S \cup \{x\})$ es factible si todas las tareas $i \in (S \cup \{x\})$ pueden ser ejecutadas en un instante anterior o igual a d_i
* Función de selección:

  Coger las tareas con mayor beneficio

Si pensamos en otras: P. ej: escoger primero aquellas que deben ser ejecutadas antes, tenemos:

i	1	2	3	4	
g_i	10	5	50	30	
d_i	2	2	3	3	

→ No entraría y sin embargo es mayor que las 2 primeras

Algoritmo

Lo complicado está en poder comprobar de forma eficiente si incluir una nueva tarea genera conflicto. Para ello se puede mantener el vector solución S ordenado (según d_i) tratando de insertar la nueva tarea en su posición correspondiente. Es fácil comprobar después si alguna de las tareas desplazadas o la propia tarea insertada se encuentra en una posición mayor a su correspondiente d_i, en ningo caso añadir la tarea NO es factible (de nuevo desde 0)

```
void factible_insertar (datos * s, int & elems, datos x)
{  // datos es una estructura con los valores
   int factible=1, i, pos                          di, gi

   for (i=elems ; i>0 && x.d < s[i-1].d
                && factible ; i--)
      if (s[i-1] > i) factible=0;

   pos=i;
   if (factible && x.d ≤ pos)   // Comprobación factibilidad
   {
      for (i=elems; i>pos; i--)
         s[i] = s[i-1];                    } inserción
      s[pos]=x;
      elems++;
   }
  return
}

void MaxBeneficio(datos * c, int n, datos * s,
                                    int & elems)
{  int i;
   ordenarporbeneficios (c, n);
   elems=0;
   for (i=0; i<n; i++)
      factible_insertar (s, elems, c[i]);
}  return
```

i	0	1	2	3
g_i	50	10	15	30
d_i	1	0	1	0

(Se adaptan los valores para que cuenten desde 0)

↓ ordenamos por beneficio

g_i	50	30	15	10
d_i	1	0	1	0

Iteración 0

S

	0
g	50
d	1

elems = 1

Iteración 1

S

	0	1
g	30	50
d	0	1

elems = 2

Iteración 2 y 3

Ninguno otro elemento puede ser insertado

S

	0	1
g	30	50
d	0	1

elems = 2

Solución óptima con beneficio 80

Ejercicio: Demostrar la optimalidad