Avanze del proyecto

Análisis y Diseño de Algoritmos(ADA)

jorge.mayna

Junio 2020

repo :https://github.com/jorgemayna/ADA_2020_proyect

1. Introducción

Los 2 arreglos de 1s y 0s que recibimos los guardamos como strings y para hallar sus respectivos bloques usamos el siguiente algoritmo:

```
BLOQUES(string s, tamaño t)
1: for i = 1 to t
2:    if s[i] = '1'
3:       first = i;
4:       while s[i] = '1'
            i++
5:       R.push_back(first, i - 1, i - first+1)
6: return R
```

El algoritmo no solo te retorna cada bloque sino que también su tamaño lo que sera usado posteriormente. Los bloques son guardados en los arreglos A[n] y B[m].

2. Algoritmo voraz

Para el algoritmo voraz se optó por el acercamiento mas sencillo, por lo cual no retorna un matching con peso mínimo, pero si retorna un matching valido. Para facilitar las cosas asumiremos que la longitud de A[n] es mayor que la de B[m](En el código en c++ si se valida esto) y que cuando usemos la notación A[i] nos estamos refiriendo al modulo del bloque i en vez de al bloque en si mismo.

```
GREEDY(Bloques A, Bloques B)                 cost    times
1: cont = 0                                  c1      1
2: for i = 1 to B.size                       c2      m + 1
3:    matching.push_back(i,i)                c3      m
```

4:	cont = cont + A[i] / B[i]	c_4	m
5:	**for** $i = B.size + 1$ to $A.size$	c_5	$n - m + 1$
6:	matching.push_back(i,B.size)	c_6	$n - m$
7:	cont = cont + A[i] / B[B.size]	c_7	$n - m$
8:	return (matching,cont)		

.
En el algoritmo anterior simplemente se separo las operaciones en 2 ciclos "for"para evitar comprobaciones innecesarias usando solo 1.
Sea $T(n)$ el tiempo de ejecución con el tamaño de A como n:
$T(n) = c_1 + c_2(m+1) + c_3(m) + c_4(m) + c_5(n-m+1) + c_6(n-m) + c_7(n-m)$
$T(n) = n(c_5 + c_6 + c_7) + m(c_2 + c_3 + c_4 - c_5 - c_6 - c_7) + c_1 + c_2 + c_5$
Si le despreciamos las diferencias entre constantes:
$T(n) = n(3c) + 3c = O(n)$

3. Recurrencia

Siendo A[n] y B[m] los vectores con los bloques de los 2 arrays de 1s y 0s originales, usaremos la notación A[i] para referirnos al modulo del bloque i en vez de al bloque en si mismo.
NOTA : Para mejor entendimiento separaremos las partes para evitar aglomeración.

$$\begin{aligned} UNO &= Min_{z=1}^{z=j-2}(OPT(i-1,z) + A[i]/\sum_{x=z+1}^{x=j} B[x]) \\ DOS &= OPT(i-1,j-1) + A[i]/B[j] \\ TRES &= Min_{z=1}^{z=i-2}(OPT(z,j-1) + (\sum_{x=z+1}^{x=j} A[x])/B[j]) \end{aligned}$$

$$OPT(i,j) = \begin{cases} A[i]/\sum_{z=1}^{z=j} B[z] & i = 1 \\ (\sum_{z=1}^{z=i} A[z])/B[j] & j = 1 \\ Min(UNO, DOS, TRES) & i > 1 \text{ and } j > 1 \end{cases}$$

4. Algoritmo recursivo

Para resolver el problema de manera recursiva se optó por el siguiente algoritmo:

OPT(i,j)
1: **if** i = 0 . .
2: cont = SUM(B[i], $1 \leq$ i $\leq$ j)
3: **Return** A[i]/cont
4: **if** j = 0

```
    cont = SUM(A[i], 1 ≤ i ≤ i)
    Return cont/[j]
uno = INF
for z = 1 to j − 1
    cont = SUM(B[x], z+1 ≤ x ≤ j)
    temp = OPT(i − 1, z) + A[i] / cont
    if temp < uno
        uno = temp
dos = INF
for z = 1 to i − 2
    cont = SUM(A[x], z+1 ≤ x ≤ i)
    temp = OPT(z, j − 1) + cont / B[j]
    if temp < dos
        dos = temp
if uno≤dos
    Return uno
Return dos
```

En el algoritmo se han juntado las parte ünoz "dos", presentadas en la recurrencia, en ünoz la parte "tres.en una parte "dos". Además, se esta suponiendo que el tamaño de A[] es mayor que el tamaño de B[] y que un bloque cualquiera se puede dividir infinitamente. Todas estas consideraciones son resueltas en el código en c++. También se ha suprimido del pseudocódigo las partes del código que se encargaban de guardar el matching en si mismo.

5. Algoritmo memoizado

Para realizar el memoizado lo único que se agrego fue una matriz nxm que guardaba los OPT ya calculados para usarlos posteriormente si se necesitaba.

```
OPT_M(i, j)
if i = 0                                                    .       .
    cont = SUM(B[i], 1 ≤ i ≤ j)
    memo[i][j] = A[i]/cont
    Return A[i]/cont
if j = 0
    cont = SUM(A[i], 1 ≤ i ≤ i)
    memo[i][j] = cont/[j]
    Return cont/[j]

uno = INF
for z = 1 to j − 1
    cont = SUM(B[x], z+1 ≤ x ≤ j)
    if memo[i-1][z] existe
        opt = memo[i-1][z]
    else
        opt = OPT_m(i − 1, z)
```

```
15:    temp = opt + A[i] / cont
16:    if temp < uno
17:       uno = temp
18:
19: dos = INF
20: for z = 1 to i − 2
21:    cont = SUM(A[x], z+1 ≤ x ≤ i)
22:    if memo[z][j-1] existe
          opt = memo[z][j-1]
23:    else
          opt = OPT_m(z, j − 1)
24:    temp = opt + cont / B[j]
25:    if temp < dos
26:       dos = temp
27: if uno≤dos
28:    memo[i][j] = uno
29:    Return uno
30: else
31:    memo[i][j] = dos
32:    Return dos
```

6. Algoritmo programación dinámica

Para el algoritmo de programación dinámica se tuvo que adaptar unas pocas cosas para que ya no funcione de manera recursiva.

```
P_DINÁMICA(BloquesA, BLoquesB)                          cost    times
 1: for i = 1 to A.size                                  .       .
 2:    cont = SUM(A[z],1 ≤ z ≤ i)                       .       .
 3:    optimos[i][0] = cont / B[0]                      .       .
 4: for i = 1 to B.size                                  .       .
 5:    cont = SUM(B[z],1 ≤ z ≤ i)                       .       .
 6:    optimos[0][i] = A[0] / cont                      .       .
 7:
 8: for j = 2 to B.size
 9:    for i = 2 to A.size
10:       uno = INF
11:       for z = 1 to j − 1
12:          cont = SUM(B[x], z+1 ≤ x ≤ j)
13:          temp = optimos[i − 1][z] + A[i] / cont
14:          if temp < uno
15:             uno = temp
16:       dos = INF
17:       for z = 1 to i − 2
18:          cont = SUM(A[x], z+1 ≤ x ≤ i)
```

```
19:          temp = optimos[z][j − 1] + cont / B[j]
20:          if temp < dos
21:             dos = temp
22:       if uno≤dos
23:          optimos[i][j] = uno
24:       else
25:          optimos[i][j] = dos
26: Return optimos[A.size][B.size]
```