Proyecto

Guillermo Franco y Alessia Yi

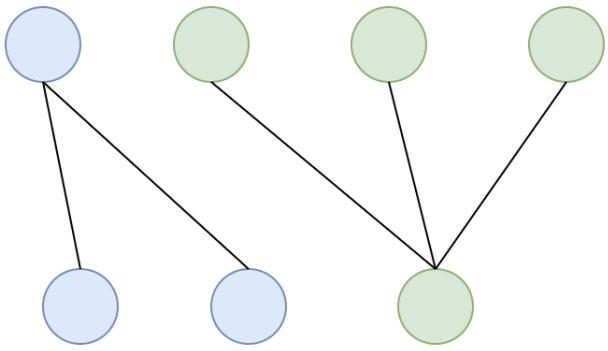
May 2020

1. Secuencias

Introducción

En este informe presentaremos tres distintos algoritmos para resolver el problema de Secuencias. Implementaremos un algoritmo voraz, otro recursivo y finalmente otro memoizado. El objetivo del algoritmo consiste en calcular el mínimo peso de un matching de dos arreglos ya existentes y devolver dicho matching. Un matching corresponde a una transformación entre de bloques de A hacia los bloques de B, tal que algunos bloques de A son divididos y otros bloques de son agrupados. El peso total de un matching es la suma de los pesos de todos los bloques que se dividen y los pesos todos los bloques que se agrupan, el peso de una división se calcula $\frac{|\text{bloque de } A|}{\text{suma de } |\text{bloques de } B_j|}$ y el peso de una agrupación de calcula $\frac{\text{suma de } |\text{bloques de } A_i|}{|\text{bloque de } B|}$.

Un ejemplo de división son los de color azul y un ejemplo de agrupación son los de color verde.



Repositorio

El repositorio con todos los algoritmos se encuentran en el siguiente link.

Pregunta 1 (Voraz)

Pseudocodigo

Primero, presentamos un subalgoritmo que calcula los bloques de los arreglos en tiempo lineal, ya que recorre dicho arreglo una sola vez. Cabe notar que este subalgoritmo se usará en todos los algoritmos futuros. Abajo de este se encuentra el algoritmo voraz principal.

Recibe: un arreglo de ceros y unos
Devuelve: un arreglo de tamaños de los bloques

```
GetBlocks(array)
 1: i = 1
 2: while i < array.size
 3:    while array[i] == 0
 4:       i = i + 1
 5:    size = 0
 6:    if array[i] == 1
 7:       while array[i] == 1
 8:          size = size + 1
 9:          i = i + 1
10:       blocks.push(size)
11: return blocks
```

Recibe: dos arreglos de ceros y unos
Devuelve: un matching entre los dos arreglos y su peso

```
GreedyMinMatching(A, B)
 1: blocksA = GetBlocks(A)
 2: blocksB = GetBlocks(B)
 3: m = blocksA.size
 4: n = blocksB.size
 5: if m > n
 6:    max_i = ratio = m/n
 7:    j = 1
 8:    for i = 1 TO m
 9:       if i ≤ max_i
10:          num = num + blocksA[i]
11:       else
12:          w = w + num/blocksB[j]
13:          num = blocksA[i]
14:          max_i = max_i + ratio
15:          j = j + 1
16:       matching.push(i, j)
```

17: $w = w + num/blocksB[j]$
18: **else**
19: $max_j = ratio = n/m$
20: $i = 1$
21: **for** $j = 1$ TO n
22: **if** $j \leq max_j$
23: $den = den + blocksB[j]$
24: **else**
25: $w = w + blocksA[i]/den$
26: $den = blocksB[j]$
27: $max_j = max_j + ratio$
28: $i = i + 1$
29: $matching.push(i, j)$
30: $w = w + blocksA[i]/den$
31: **return** $matching, w$

Analisis

Dejando de lado el cálculo de los bloques de cada arreglo, el tiempo de ejecución del algoritmo propuesto es de $O(max\{m, n\})$. Esto se debe al condicional de la linea 5. Si m, la cantidad de bloques del arreglo A, es mayor, se itera asignándole un solo *matching* a cada bloque de A. En el caso contrario, se hace lo mismo pero con los bloques de B. El calculo del peso del *matching* w se hace durante este proceso, por lo que solo le aumenta tiempo constante a la iteración.

Implementación

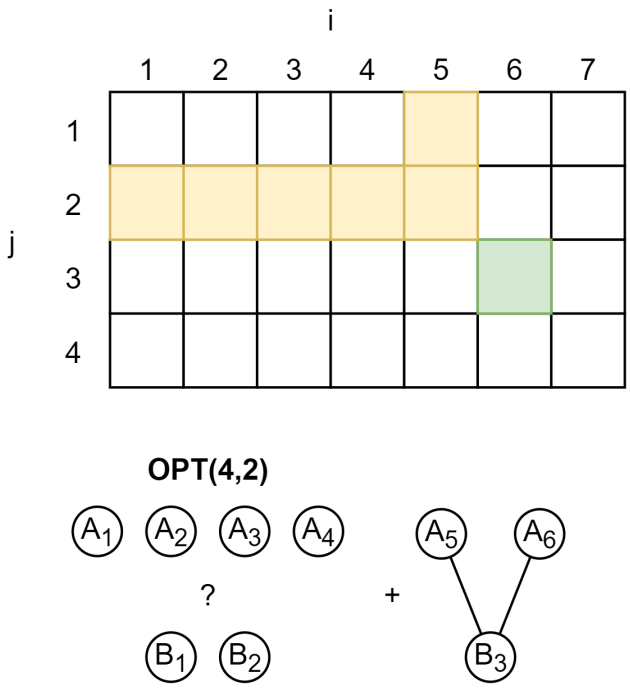
La implementación del algoritmo en C++ se puede encontrar en el siguiente link.

Pregunta 2 (Recurrencia)

$$
\begin{aligned}
OPT(i,j) &= \begin{cases} A_1/sum(B_1..B_j) & \text{si } i = 1 \\ sum(A_1..A_i)/B_1 & \text{si } j = 1 \\ OPT(i,j) = min(Agrupar(i,j), Dividir(i,j)) & \end{cases} \\
Agrupar(i,j) &= min(OPT(x, j-1) + sum(A_{x+1}..A_i)/B_j) : \quad x \in \{1..i-1\} \\
Dividir(i,j) &= min(OPT(i-1, y) + A_i/sum(B_{y+1}..B_j)) : \quad y \in \{1..j-1\}
\end{aligned}
$$

La siguiente imágen representa cómo funciona la recurrencia. Para calcular el OPT(6,3), mostrado de color verde, se necesita evaluar todos los de color amarillo. La subrutina Agrupar consta de la fila evaluada, mientras que Dividir evalua la columna.

Por ejemplo, cuando se calcula el OPT(6,3), una de las combinaciones a evaluarse es (4,2). Se calcula dicho OPT, y se le suma el peso del matching restante, como se ilustra en la figura a continuación.



Pregunta 3 (Recursivo)

Pseudocódigo

Este primer algoritmo es el principal. Transforma los arreglos a bloques y llama a la subrutina recursiva con los valores obtenidos. Esta subrutina, mostrada a continuación del primero, es la transformación de la fórmula de recurrencia presentada previamente a pseudocódigo.

Recibe: dos arreglos de ceros y unos
Devuelve: un matching de peso mínimo entre los dos arreglos, y el respectivo peso

RecursiveMinMatching (A, B)
1: $blocksA = \text{GetBlocks}(A)$
2: $blocksB = \text{GetBlocks}(B)$
3: $m = blocksA.size$
4: $n = blocksB.size$
5: **return** RecursiveMinMatchingRec$(blocksA, blocksB, m, n)$

Recibe: dos arreglos de tamaños de bloques, y la cantidad de bloques a usar
Devuelve: un matching de peso mínimo entre los bloques usados y su peso
RecursiveMinMatchingRec $(blocksA, blocksB, i, j)$

```
1: if i = 1
2:    w = blocksA[1]/(sum(blocksB[1]..blocksB[j])
3:    matching.push((1, 1)..(1, j))
4:    return matching, w
5: if j = 1
6:    w = sum(blocksA[1]..blocksA[i])/blocksB[1]
7:    matching.push((1, 1)..(i, 1))
8:    return matching, w
9: w = INF
10: for x=1 TO i-1
11:    group, w_g = RecursiveMinMatchingRec(blocksA, blocksB, x, j − 1)
12:    w_g = w_g + sum(blocksA[x + 1]..blocksA[i])/blocksB[j])
13:    group.push((x + 1, j)..(i, j))
14:    if w_g < w
15:       w = w_g
16:       matching = group
17: for y=1 TO j-1
18:    divide, w_d = RecursiveMinMatchingRec(blocksA, blocksB, i − 1, y)
19:    w_d = w_d + blocksA[i]/sum(blocksB[y + 1]..blocksB[j])
20:    divide.push((i, y + 1)..(i, j))
21:    if w_d < w
22:       w = w_d
23:       matching = divide
24: return matching, w
```

Análisis

Como la llamada recursiva siempre reduce los valores de m y n por lo menos en 1, el árbol de recursión tendrá, a lo máximo, altura de $min(m, n)$. Además, para cualquier pares de m y n en los que ambos son mayores a 1, siempre llamarán dos veces al la recursión con valores $m - 1$ y $n - 1$, dejando de lado todas las otras llamadas. Esto quiere decir que, solo tomando una pequeña fracción de las llamadas recursivas, estas se doblan con cada nivel del arbol. Así, tenemos una cota muy inferior de $\Omega(2^{min(m,n)})$.

Si evaluamos la recursión de forma más completa, una llamada con valores m y n resultan en $m + n - 2$ llamadas recursivas, ya que *Agrupar* y *Dividir* llaman a la recursión $m - 1$ y $n - 1$ respectivamente. Así, podemos afirmar que la cantidad de llamadas recursivas depende más del valor máximo entre m y n. Por lo tanto, es más preciso decir que la recursión tiene un tiempo de ejecución de $\Omega(max(m, n)^{min(m,n)})$, y eso es si el algoritmo en sí toma tiempo constante, lo cual está muy alejado de la realidad.

Implementación

La implementación del algoritmo en C++ se puede encontrar en el siguiente link.

Pregunta 4 (Memoizado)

Pseudocódigo

Este primer algoritmo es el principal. Transforma los arreglos a bloques, crea los arreglos donde se guardarán los pesos con sus respectivos matching, llama a la subrutina recursiva. Esta subrutina, mostrada a continuación del primero, es el algoritmo recursivo transformado a memoizado. Los pesos y sus respectivos matching solo se cáculan si es que no han sido obtenidos previamente.

Recibe: dos arreglos de ceros y unos
Devuelve: un matching de peso mínimo entre los dos arreglos, y el respectivo peso

```
MemoMinMatching (A, B)
1: blocksA = GetBlocks(A)
2: blocksB = GetBlocks(B)
3: m = blocksA.size
4: n = blocksB.size
5: matching[m][n]
6: memo[m][n] = INF
7: return MemoMinMatchingRec(blocksA, blocksB, m, n, matching, memo)
```

Recibe: dos arreglos de tamaños de bloques, la cantidad de bloques a usar, y la memoria
Devuelve: un matching entre los bloques usados de peso mínimo

```
MemoMinMatchingRec (blocksA, blocksB, i, j, matching, memo)
1: if memo[i][j] ≠ INF
2:    return matching[i][j], memo[i][j]
3: if i = 1
4:    w = blocksA[1]/(sum(blocksB[1]..blocksB[j])
5:    matching[i][j].push((1, 1)..(1, j))
6:    memo[i][j] = w
7:    return matching[i][j], w
8: if j = 1
9:    w = sum(blocksA[1]..blocksA[i])/blocksB[1]
10:   matching[i][j].push((1, 1)..(i, 1))
11:   memo[i][j] = w
12:   return matching[i][j], w
13: w = INF
14: for x=1 TO i-1
15:   group, w_g = MemoMinMatchingRec(blocksA, blocksB, x, j − 1, matching, memo)
16:   w_g = w_g + sum(blocksA[x + 1]..blocksA[i])/blocksB[j])
```

```
    group.push((x + 1, j)..(i, j))
    if w_g < w
        w = w_g
        matching[i][j] = group
for y=1 TO j-1
    divide, w_d = MemoMinMatchingRec(blocksA, blocksB, i − 1, y, matching, memo)

    w_d = w_d + blocksA[i]/sum(blocksB[y + 1]..blocksB[j])
    divide.push((i, y + 1)..(i, j))
    if w_d < w
        w = w_d
        matching[i][j] = divide
memo[i][j] = w
return matching[i][j], w
```

Análisis

El tiempo de complejidad del algoritmo es $O(mn)$. Se crea una tabla *memo* de tamaño $m * n$ para almacenar los resultados calculados por primera vez. Es decir, la recurrencia solo demora $m * n$ llamadas en llenar la tabla, y después todas las llamadas son constantes.

Implementación

La implementación del algoritmo en C++ se puede encontrar en el siguiente link.