Análisis y Diseño de Algoritmos

Proyecto Final

Benjamín Díaz García
benjamin.diaz@utec.edu.pe

Gabriel Spranger Rojas
gabriel.spranger@utec.edu.pe

Rodrigo Céspedes Velásquez
rodrigo.cespedes@utec.edu.pe

Profesor:
Ph.D. Juan Gabriel Gutiérrez Alva

July 25, 2020

Introducción

En el presente trabajo se presentarán, explicarán y demostrarán algoritmos ideados por nosotros para resolver diferentes problemas propuestos.
Cada problema será resuelto a través de distintos tipos de algoritmos (Voraz, Recursivo, Memoizado y de Programación Dinámica.)

A lo largo del trabajo se seguirá una estructura predefinida.
Primero, se presentará el problema, tipo de algoritmo y lo que es requerido del algoritmo, así como sus entradas y salidas. Después se exhibirá el pseudocódigo correspondiente a nuestra solución. Finalmente, se expondrá el análisis de tiempo para dicha respuesta, además de su demostración. En algunos casos también se pondrá la recurrencia correspondiente (si es pertinente).

Puede que para algunas preguntas existan formatos diferentes, pues en algunos casos se recurre a distintos métodos para evidenciar o demostrar alguna característica de nuestra respuesta, pero en general casi todas las preguntas siguen la misma estructura.

1. Algoritmo Voraz

Analice, diseñe e implemente un algoritmo voraz con complejidad lineal para el problema MIN-MATCHING. Su algoritmo no necesariamente debe encontrar el matching de peso mínimo.

Require: Dos arreglos A y B de ceros y unos de tamaño p con n y m bloques, respectivamente.
Ensure: Un matching entre A y B, no necesariamente óptimo, y su peso.
MIN-MATCHING-GREEDY(A', B')**:**

1: Sea A' el arreglo con los bloques de A con n bloques.
2: Sea B' el arreglo con los bloques de B con m bloques
3: **while** $i < m - 1$ **and** $j < n - 1$ **do**
4: **if** $A'[i] == B'[j]$ **then**
5: $auxA'$.pushback(i)
6: $auxB'$.pushback(j)
7: R.pushback($pair(auxA', auxB')$)
8: $sumavalor$ += $A'[i]$ / $B'[j]$
9: $i++$
10: $j++$
11: **end if**
12: **else if** $A'[i] > B'[j]$ **then**
13: $auxA$ .pushback(i)
14: $valA' = A'[i]$
15: **while** $(valA' > valB')$ **and** $(j < n - 1)$ **do**
16: $auxB'$.pushback(j)
17: $valB'$ += $B'[j]$
18: $j++$
19: **end while**
20: R.pushback($pair(auxA', auxB')$)
21: $sumavalor$ += $valA$ / $valB'$
22: $i++$
23: **else**
24: $auxB'$.pushback(j)
25: $valB' = B'[j]$
26:
27: **while** $(valA' < valB')$ **and** $(i < m - 1)$ **do**

28: $auxA'$.pushback(i)
29: $valA'$ += $A'[i]$
30: $i++$
31: **end while**
32: R.pushback($pair(auxA', auxB')$)
33: $sumavalor$ += $valA'$ / $valB'$
34: $j++$
35: $valA' = 0$
36: $valB' = 0$
37: $auxA'$.clear()
38: $auxB'$.clear()
39: **end while**
40:
41: $auxA'$.pushback($i, (i+1) \dots (m-1)$)
42: $valA'$ += $(A'[i], A'[i+1] \dots A'[m-1])$
43:
44: $auxB'$.pushback($j, (j+1) \dots (n-1)$)
45: $valB'$ += $(B'[j], B'[j+1] \dots B[n-1])$
46:
47: R.pushback($pair(auxA', auxB')$)
48: $sumavalor$ += $valA'$ / $valB'$
49:
50: **return** $pair(sumavalor, R)$

Análisis de Tiempo

Las veces seran analizadas en el peor de los casos. Por temas de practicidad utilizaremos la letra c como la constante de mayor valor posible en el contexto donde se utiliza.

Linea 3 a 11:
Tiempo: c
Veces: min(m-x, n-y), ($0 \leq$ x,y $\leq$ m-1,n-1)

Linea 12 a 14:
Tiempo: c
Veces: min(m-x, n-y)

Linea 15 a 19:
Tiempo: c
Veces: x

Linea 20 a 22:
Tiempo: c
Veces: min(m-x, n-y)

Linea 23 a 26:
Tiempo: c
Veces: min(m-x, n-y)

Linea 27 a 31:
Tiempo: c
Veces: y

Linea 32 a 39:
Tiempo: c
Veces: min(m-x, n-y)

Linea 41:
Tiempo: m-(m-x)
Veces: 1

Linea 42:
Tiempo: m-(m-x)
Veces: 1

Linea 44:
Tiempo: n-(n-y)
Veces: 1

Linea 45:
Tiempo: n-(n-y)
Veces: 1

Linea 47 a 50:
Tiempo: c

Veces: 1

Entonces tenemos que:

c*min(m-x,n-y) + c*min(m-x,n-y) + c*x + c*min(m-x,n-y) + c*min(m-x,n-y) +c*y + c*min(m-x,n-y) + c*(m-(m-x)) + c*(m-(m-x)) + c*(n-(n-y)) + c*(n-(n-y)) + c

Es equivalente a:

c*min(m-x,n-y) + c*x + c*y + c*(m-(m-x)) + c*(n-(n-y)) + c

Es equivalente a:

c*min(m-x,n-y) + c*x + c*y + c*m - c*(m-x) + c*n - c*(n-y) + c

Es equivalente a:

c*min(m-x,n-y) + c*x + c*y + c*m - c*m + c*x + c*n - c*n + c*y + c

Es equivalente a:

c*min(m-x,n-y) + c*x + c*y + c*x + c*y + c

Se sabe que min(m-x,n-y) $\leq$ max(m,n), entonces esta expresión es mayor o igual

c*max(m,n) + c*m + c*n + c*m + c*n + c

Es equivalente a:

c*max(m,n) + c*m + c*n + c

Sabemos que m $\leq$ max(m,n) y que n $\leq$ max(m,n), entonces la siguiente expresión es mayor o igual

c*max(m,n) + c*max(m,n) + c*max(m,n) + c*max(m,n)

Lo cual es equivalente a

c*max(m,n)

Y sabemos que c*max(m,n) $\leq$ O(max(m,n))

2. Recurrencia

Plantee una recurrencia para $OPT(i, j)$.

$$OPT(i,j) = \begin{cases} \frac{A_1}{B_1} & \text{si } i = 1 \text{ y } j = 1 \\ \frac{A_1+A_2+...+A_i}{B_1} & \text{si } j = 1 \text{ y } i > 1 \\ \frac{A_1}{B_1+B_2+...+B_j} & \text{si } i = 1 \text{ y } j > 1 \\ \min \left\{ \begin{array}{l} \min_{k=j-1}^{1}\{ OPT(i-1,k) + \frac{A_i}{B_{k+1}+...+B_j} \}, \\ \min_{k=1}^{i-1}\{ OPT(k,j-1) + \frac{A_{k+1}+...+A_i}{B_j} \} \end{array} \right\} & \text{caso contrario} \end{cases}$$

3. Recursivo

Analice, diseñe e implemente un algoritmo recursivo con complejidad exponencial para el problema MIN-MATCHING. Su algoritmo deberá encontrar el matching del peso mínimo.

Require: Dos arreglos A y B de ceros y unos, de tamaño p con n y m bloques, respectivamente.
Ensure: Un matching entre A y B, de peso mínimo, y su peso.

MIN-MATCHING(A, B)**:**
1: Sea A' el arreglo con los bloques de A con n bloques.
2: Sea B' el arreglo con los bloques de B con m bloques
3: Sea X el vector de las conexiones que se van a realizar entre A' y B'.
4: Sea *Respuesta* un $pair < vector, float >$ que contiene X y el peso.
5: **if** $n == 1$ **and** $m == 1$ **then**
6: Respuesta.first.emplace_back(A'_1, B'_1)
7: Respuesta.second $= \dfrac{A'_1}{B'_1}$
8: **return** Respuesta
9: **end if**
10:
11: **if** $m == 1$ **then**
12: Respuesta.first.emplace_back$(A'_1 \cup A'_2 \cup \ldots \cup A'_n, B'_1)$
13: Respuesta.second $= \dfrac{A'_1 + A'_2 + \ldots + A'_n}{B'_1}$
14: **return** Respuesta
15: **end if**
16:
17: **if** $n == 1$ **then**
18: Respuesta.first.emplace_back$(A'_1, B'_1 \cup B'_2 \cup \ldots \cup B'_m)$
19: Respuesta.second $= \dfrac{A'_1}{B'_1 + B'_2 + \ldots + B'_m}$
20: **return** Respuesta
21: **end if**
22:
23: $min \leftarrow \infty$
24: $A_* = A \setminus A'_n$ {Le quitamos el último bloque a A}

25: **for** k = m - 1 **downto** 1 **do**
26: $B_* = B'_{1...k}$ {Agarramos los primeros k bloques}
27: Respuesta' $\leftarrow$ MIN-MATCHING(A_*, B_*)
28: Respuesta'.second $\leftarrow$ Respuesta'.second $+ \frac{A'_n}{B_{k+1}+...+B_m}$
29: **if** $min >$ Respuesta'.second **then**
30: Respuesta.first.clear()
31: Respuesta.first.emplace_back$(A'_n, B_{k+1} \cup \ldots \cup B_m)$
32: Respuesta.first = Respuesta.first $\cup$ Respuesta'.first
33: Respuesta.second = Respuesta'.second
34: **end if**
35: **end for**
36:
37: $B_* = B \setminus B'_m$ {Le quitamos el último bloque a B}
38: **for** k = 1 **to** n - 1 **do**
39: $A_* = A'_{1...k}$ {Agarramos los primeros k bloques}
40: Respuesta' $\leftarrow$ MIN-MATCHING(A_*, B_*)
41: Respuesta'.second $\leftarrow$ Respuesta'.second $+ \frac{A'_{k+1}+...+A'_n}{B'_m}$
42: **if** $min >$ Respuesta'.second **then**
43: Respuesta.first.clear()
44: Respuesta.first.emplace_back$(A'_{k+1} \cup \ldots \cup A'_n, B'_m)$
45: Respuesta.first = Respuesta.first $\cup$ Respuesta'.first
46: Respuesta.second = Respuesta'.second
47: **end if**
48: **end for**
49:
50: **return** Respuesta

Análisis de Tiempo

El tiempo del algoritmo viene dado por la siguiente relación de recurrencia:

$$T(n,m) = \begin{cases} m + k_1 & \text{si } n = 1 \text{ y } m > 1 \\ n + k_2 & \text{si } m = 1 \text{ y } n > 1 \\ k_3 & \text{si } m = 1 \text{ y } n = 1 \\ \sum_{k=2}^{m-1} T(n-1,k) + \sum_{k=2}^{n-1} T(k,m-1) & \text{caso contrario} \end{cases}$$

Ahora demostremos que $T(n,m) = \Omega(2^{\max\{n,m\}})$.

En el caso base tenemos que $n = 1$ y $m = 1$:

$$k_3 \geq c\ 2^{\max\{1,1\}}$$

Si $c = \dfrac{k_3}{2}$ tenemos que:

$$k_3 \geq k_3$$

Por lo tanto, el caso base cumple. Ahora veamos el caso inductivo:

$$T(n,m) \geq c\ 2^{\max\{n,m\}}$$

T(n,m) es:

$$T(n,m) = \sum_{k=2}^{m-1} T(n-1,k) + \sum_{k=2}^{n-1} T(k,m-1)$$

Le quitamos una sumatoria, la desigualdad se mantiene:

$$T(n,m) \geq \sum_{k=2}^{m-1} T(n-1,k)$$

Agarramos el último término de la sumatoria:

$$T(n,m) \geq T(n-1,m-1)$$

Por hipótesis inductiva tenemos que:

$$T(n-1, m-1) \geq c\ 2^{\max\{n-1, m-1\}}$$

Le sumamos 1 a n y m:

$$T(n, m) \geq c\ 2^{\max\{n,m\}-1}$$

Por lo tanto, si $c = \frac{1}{2}$:

$$T(n, m) = \Omega(2^{\max\{n,m\}})$$

4. Memoizado

Analice, diseñe e implemente un algoritmo memoizado para el problema MIN-MATCHING. Su algoritmo deberá encontrar el matching de peso minimo.

Require: Dos arreglos A y B de ceros y unos, de tamaño p con n y m bloques, respectivamente.
Ensure: Un matching entre A y B, de peso mínimo, y su peso.

MIN-MATCHING(A, B)**:**
1: Sea A' el arreglo con los bloques de A con n bloques.
2: Sea B' el arreglo con los bloques de B con m bloques
3: Sea X el vector de las conexiones que se van a realizar entre A' y B'.
4: Sea *Respuesta* un $pair < vector, float >$ que contiene X y el peso.
5: Sea M una matriz declarada globalmente que contiene las *Respuestas*.
6: **if** $M[n][m]$!= ∞ **then**
7: **return** $M[n][m]$
8: **end if**
9:
10: **if** $n == 1$ **and** $m == 1$ **then**
11: **if** $M[1][1] == \infty$ **then**
12: Respuesta'.first.emplace_back(0, 0)
13: Respuesta'.second = $\frac{A'_1}{B'_1}$
14: $M[1][1]$ = Respuesta'
15: **end if**
16: **return** $M[1][1]$
17: **end if**
18:
19: **if** $m == 1$ **then**
20: **if** $M[n][1] == \infty$ **then**
21: Respuesta'.first.emplace_back($A'_1 \cup A'_2 \cup \ldots \cup A'_n, B'_1$)
22: Respuesta'.second = $\frac{A'_1 + A'_2 + \ldots + A'_n}{B'_1}$
23: $M[n][1]$ = Respuesta'
24: **end if**

25: **return** $M[n][1]$
26: **end if**
27:
28: **if** $n == 1$ **then**
29: **if** $M[1][n] == \infty$ **then**
30: Respuesta'.first.emplace_back($A'_1, B'_1 \cup B'_2 \cup \ldots \cup B'_m$)
31: Respuesta'.second = $\frac{A'_1}{B'_1 \cup B'_2 \cup \ldots \cup B'_m}$
32: $M[1][n]$ = Respuesta'
33: **end if**
34: **return** $M[1][n]$
35: **end if**
36:
37: $min \leftarrow \infty$
38: $A_* = A \setminus A'_n$ {Le quitamos el último bloque a A}
39: **for** k = m - 1 **downto** 1 **do**
40: $B_* = B'_{1\ldots k}$ {Agarramos los primeros k bloques}
41: Sea n_* y m_* el número de bloques en A_* y B_*, respectivamente.
42: **if** $M[n_*][m_*]$!= ∞ **then**
43: Respuesta' = $M[n_*][m_*]$
44: Respuesta'.first = Respuesta'.first $\cup$ $(A_n, B_{k+1} \cup \ldots \cup B_m)$
45: Respuesta'.second += $\frac{A_n}{B_{k+1}+\ldots+B_m}$
46: **else**
47: Respuesta' = MIN-MATCHING(A_*, B_*)
48: Respuesta'.first = Respuesta'.first $\cup$ $(A_n, B_{k+1} \cup \ldots \cup B_m)$
49: Respuesta'.second += $\frac{A_n}{B_{k+1}+\ldots+B_m}$
50: **end if**
51: **if** $min >$ Respuesta'.second **then**
52: Respuesta = Respuesta'
53: **end if**
54: **end for**
55:
56: $B_* = B \setminus B'_m$ {Le quitamos el último bloque a B}
57: **for** k = 1 **to** n - 1 **do**
58: $A_* = A'_{1\ldots k}$ {Agarramos los primeros k bloques}
59: Sea n_* y m_* el número de bloques en A_* y B_*, respectivamente.
60: **if** $M[n_*][m_*]$!= ∞ **then**

61: Respuesta' = $M[n_*][m_*]$
62: Respuesta'.first = Respuesta'.first $\cup$ $(A_{k+1} \cup \ldots \cup A_n, B_m)$
63: Respuesta'.second += $\frac{A_n}{B_{k+1}+\ldots+B_m}$
64: **else**
65: Respuesta' = MIN-MATCHING(A_*, B_*)
66: Respuesta'.first = Respuesta'.first $\cup$ $(A_{k+1} \cup \ldots \cup A_n, B_m)$
67: Respuesta'.second += $\frac{A_{k+1}+\ldots+A_n}{B_m}$
68: **end if**
69: **if** $min >$ Respuesta'.second **then**
70: Respuesta = Respuesta'
71: **end if**
72: **end for**
73:
74: **return** Respuesta

Análisis de Tiempo

En este caso, cada vez que calculamos el peso mínimo de una combinación en especial, la guardamos en su lugar correspondiente en la matriz de memoización para que siguientes llamadas recursivas con los mismos parámetros solo agarren el valor de la matriz, de tal manera, tenemos un acceso en $O(1)$ para resultados memoizados. Como cada llamada recursiva siempre estará dentro de los límites n y m de la matriz de memoización, cada celda será calculada como máximo, una vez. Por lo tanto, tenemos que en el peor caso, llenaremos todas las celdas de la matriz, teniendo un total de n x m accesos a la matriz. Por ello, podemos concluir que el algoritmo memoizado corre en $O(mn)$.

5. Programación Dinámica

Analice, diseñe e implemente un algoritmo de programación dinámica para el problema Min-Matching. Su algoritmo deberá encontrar el matching de peso mínimo.

Require: Dos arreglos A y B de ceros y unos, de tamaño p con n y m bloques, respectivamente.

Ensure: Un matching entre A y B, de peso mínimo, y su peso.

MIN-MATCHING(A, B)**:**

Sea A' el arreglo con los bloques de A con n bloques.
Sea B' el arreglo con los bloques de B con m bloques
Sea X el vector de las conexiones que se van a realizar entre A' y B'.
Sea $Respuesta_i$ un $pair < vector, float >$ que contiene X y el peso.
Sea M una matriz declarada globalmente que contiene las *Respuestas*.
$sum \leftarrow 0$
for i = 1 **to** n **do**
    Respuesta'.first.emplace_back($A'_1 \cup \ldots \cup A'_i$, B'_1)
    $sum \leftarrow sum + A_i$
    Respuesta'.second $\leftarrow sum$
    M[i][1] = Respuesta'
end for

$sum \leftarrow 0$
for j = 1 **to** m **do**
    Respuesta'.first.emplace_back(A'_1, $B'_1 \cup \ldots \cup B'_j$)
    $sum \leftarrow sum + B_j$
    Respuesta'.second $\leftarrow sum$
    M[1][j] = Respuesta'
end for

$min_1 \leftarrow \infty$
$min_2 \leftarrow \infty$

$sum_1 \leftarrow 0$
$sum_2 \leftarrow 0$

$weight_1 \leftarrow 0$
$weight_2 \leftarrow 0$
$conexiones_1 \leftarrow \emptyset$
$conexiones_2 \leftarrow \emptyset$

for i = 2 **to** n **do**
    for j = 2 **to** m **do**
        for k = j-1 **to** 1 **do**

36: $conexiones_1 \leftarrow \emptyset$
37: **for** a = k+1 **to** j **do**
38: $sum_1 \leftarrow B_a$
39: $conexiones_1 \leftarrow conexiones_1 \cup B_a$
40: **end for**
41: $conexiones_1 \leftarrow M[i-1][k].first \cup conexiones_1$
42: $weight_1 \leftarrow M[i-1][k].first + \frac{A_i}{sum_1}$
43: **if** $weight_1 < min_1$ **then**
44: $Respuesta_1.first \leftarrow weight_1$
45: $Respuesta_1.second \leftarrow conexiones_1$
46: **end if**
47: **end for**
48: **for** k = 1 **to** i-1 **do**
49: $conexiones_2 \leftarrow \emptyset$
50: **for** a = k+1 **to** i **do**
51: $sum_2 \leftarrow A_a$
52: $conexiones_2 \leftarrow conexiones_2 \cup A_a$
53: **end for**
54: $conexiones_2 \leftarrow M[k][j-1].first \cup conexiones_2$
55: $weight_2 \leftarrow M[k][j-1].second + \frac{sum_2}{B_j}$
56: **if** $weight_2 < min_2$ **then**
57: $Respuesta_2.first \leftarrow weight_2$
58: $Respuesta_2.second \leftarrow conexiones_2$
59: **end if**
60: **end for**
61:
62: **if** $min_1 < min_2$ **then**
63: $M[i][j] \leftarrow Respuesta_1$
64: **else**
65: $M[i][j] \leftarrow Respuesta_2$
66: **end if**
67: **end for**
68: **end for**
69: **return** $M[n][m]$

Análisis de Tiempo

$$T(n,m) = \begin{cases} m + k_1 & \text{si } n = 1 \text{ y } m > 1 \\ n + k_2 & \text{si } m = 1 \text{ y } n > 1 \\ k_3 & \text{si } m = 1 \text{ y } n = 1 \\ m * m * n * k_4 + n + m + k_5 & \text{caso contrario} \end{cases}$$

Explicaremos el tiempo de ejecución por partes. Primero, el m * m * n * k_4 viene del for anidado que empieza en la línea 31 del pseudocódigo. Se hacen m * $m - 1$ iteraciones n veces (en verdad, se hacen m * $m - 1$ iteraciones, solo 1 vez, ya que luego se le resta 2 a m, luego 3, luego 4 y así sucesivamente hasta 1, sin embargo, para sacar el *upper-bound* esa suposición es válida), por eso se multiplica y además se multiplica un k_4, ya que en cada iteración se hace trabajo que depende del i y el j de cada iteración. Si bien es cierto que hay bucles for dentro, lo que nos importa es el for anidado i y j, ya que esos dependen de m y n, lo cual son nuestros inputs. Por ello, decidimos llamar el trabajo que se hace en cada iteración del for anidado k_4. Segundo, el $m + n + k_5$ viene de los dos primeros bucles for en el comienzo del pseudocódigo. El primero itera n veces y el segundo itera n veces, por lo tanto, es $m + n$. Además, en cada iteración de ambos bucles, se hace trabajo "constante", el cual consta de llenar los casos base en la matriz de programación dinámica, por lo que se le pone a ese trabajo conjunto de ambos bucles, k_5, el cual es igual a k_1 (trabajo constante del primero bucle) + k_2 (trabajo constante del segundo bucle. A continuación hacemos la demostración formal de que el algoritmo corre en $\mathcal{O}(m^2 n)$:

$$m^2 n k_4 + n + m + k_5 \leq c m^2 n$$

Obviemos las constantes ya que estamos acotando superiormente:

$$m^2 n + n + m \leq c m^2 n$$

Digamos que $c = 3$, entonces:

$$m^2 n + n + m \leq 3 m^2 n$$

Lo cual es igual a:

$$m^2n + n + m \leq m^2n + m^2n + m^2n$$

Lo cual demuestra que nuestro algoritmo corre en $\mathcal{O}(m^2n)$.

6. Transformación Voraz

Analice, diseñe e implemente un algoritmo voraz con complejidad cuadrática para el problema MIN-TRANSFORMACION. Su algoritmo no necesariamente deberá encontrar la transformación de peso mínimo. Debe usar como subrutina al algoritmo implementado en la Pregunta 1.

Require: Dos matrices A y B de ceros y unos de tamaño p x q.

Ensure: Una transformación óptima entre A y B, y su peso.
Tiempo de ejecución del algoritmo: $\mathcal{O}(pq)$.

MIN-TRANSFORMATION-GREEDY(A', B')**:**

1: Sea A' la matriz donde cada fila tiene los bloques de cada fila de A, $\forall_{1 \leq i \leq p}$.
2: Sea B' la matriz donde cada fila tiene los bloques de cada fila de B, $\forall_{1 \leq i \leq p}$.
3: Sea *respuesta* el vector de pairs que almacena en el primer elemento el matching para las correspondientes filas i de A' y B' y en el segundo elemento el peso de dicho matching. Recordemos que el algoritmo MIN-MATCHING-GREEDY retorna un pair. En el primer elemento del pair se encuentra el matching y en el segundo elemento, su peso.
4: **for** $i = 1$ **to** $A'.size()$ **do**
5: *respuesta* .pushback(MIN-MATCHING-GREEDY$(A'[i], B'[i])$)
6: **end for**
7: **return** respuesta

Análisis de Tiempo

Sabemos que el costo de **Min-Matching-Greedy** es de $\mathcal{O}(max(m,n))$. Donde n es el número de bloques de A y m el número de bloques de B, donde A y B tienen tamaño q. Observe que si **Min-Matching-Greedy** corre en $\mathcal{O}(max(m,n))$, entonces también corre en $\mathcal{O}(q)$, ya que siempre el número de bloques será menor al número de bits en el arreglo. Entonces, como **Min-Matching-Greedy** corre en $\mathcal{O}(q)$ y llamamos a la función **Min-Matching-Greddy** p veces, podemos afirmar que el algoritmo **Min-Transformacion-Greedy** corre en $\mathcal{O}(pq)$.

7. Transformación Programación Dinámica

Analice, diseñe e implemente un algoritmo de programación dinámica con complejidad cúbica para el problema MIN-TRANSFORMACION. Su algoritmo deberá devolver una transformación de peso mínimo. Debe usar como subrutina al algoritmo implementado en el Pregunta 5.

Tiempo de ejecución del algoritmo: $\mathcal{O}(pq^3)$.

Require: Dos matrices A y B de ceros y unos de tamaño p x q.
Ensure: Una transformación óptima entre A y B, y su peso.

MIN-TRANSFORMACION(A, B)**:**

1: Sea A' la matriz donde cada fila tiene los bloques de cada fila de A, $\forall_{1 \leq i \leq p}$.
2: Sea B' la matriz donde cada fila tiene los bloques de cada fila de B, $\forall_{1 \leq i \leq p}$.
3: Sea *peso* la variable que va a guardar el peso del Min-Matching entre A y B.
4: Sea M una matriz donde cada fila i de M contiene el min-matching entre la fila i de A y B, $\forall_{1 \leq i \leq p}$. Es decir, cada fila i de M es un array de pairs que representan el min-matching entre la fila i de A y B.
5:
6: $peso \leftarrow 0$
7:

8: **for** i = 1 **to** p **do**
9: Sea A'_i la i-ésima fila de A'
10: Sea B'_i la i-ésima fila de B'
11: $pair \leftarrow$ **Min-Matching**(A'$_i$, B'_i)
12: $peso \leftarrow$ peso + $pair.second$
13: $M[i] \leftarrow pair.first$
14: **end for**
15:
16: Sea *respuesta* un pair tal que:
17: $respuesta.first \leftarrow M$
18: $respuesta.second \leftarrow peso$
19:
20: **return** *respuesta*

Análisis de Tiempo

Sabemos que el costo de **Min-Matching** es de $\mathcal{O}(m^2n)$. Donde n es el número de bloques de A y m el número de bloques de B, donde A y B tienen tamaño q. Observe que si **Min-Matching** corre en $\mathcal{O}(m^2n)$, entonces también corre en $\mathcal{O}(q^3)$, ya que siempre el número de bloques será menor al número de bits en el arreglo. Entonces, como **Min-Matching** corre en $\mathcal{O}(q^3)$ y llamamos a la función **Min-Matching** p veces, podemos afirmar que el algoritmo **Min-Transformacion** corre en $\mathcal{O}(pq^3)$.

8. Lectura de imágenes

Implementar una función de lectura de imágenes. Deberá recibir como entrada una imagen y devolver una matriz de 0s y 1s que codifica a la imagen leída. Esta función deberá ser lo suficientemente flexible (recibir tres parámetros adicionales) para poder usar uno u otro método de transformación. Así también, debería ser flexible para decidir cual es el umbral que decide si se transforma a 0 o a 1.

Esta función llamada TRANSFORM, retorna la imagen transformada representada por una matriz de 0s y 1s, se puede encontrar en nuestro código dentro de la clase MATRIXTRANSFORMER, la cual sirve como una asbtracción

para lidiar con todo lo relacionado a la transformación de la imagen, la cual inicialmente es representada por pixeles que tienen tres componentes (RGB), y despues de ser transformada, se convierte en una matriz de 0s y 1s, un cero o un uno por cada pixel. Usamos tres métodos para decidir si un pixel con los componentes **R**, **G** y **B**, debe convertirse a **0** o a **1**. Estos métodos son los siguientes: 601, 709 y 240. Cada método representa una fórmula que calcula el *brightness* en función de los componentes RGB. La fórmula se ve de la siguiente manera:

$$Y = c_1R + c_2G + c_3B$$

Donde Y representa el *brightness* de ese pixel y R, G y B representan el valor de cada componente del pixel. Cada método tiene distintas constantes o coeficientes (c_1, c_2 y c_3). Si el Y era menor que un *umbral* que se le pasa como parámetro al constructor de MatrixTransformer, entonces el pixel podía ser representado por un **1**, ya que el *brightness* es "bajo" y por lo tanto el pixel está más cerca de negro. Si el Y era mayor o igual que el *umbral*, entonces el dicho pixel podía ser representado por un **0**, ya que el *brightness* es "alto" y por lo tanto el pixel está más cerca de blanco. Repetimos esta operación para cada pixel en la imagen, insertando el 0 o 1 en la nueva matriz. La razón por la que ponemos "alto" y "bajo" entre comillas, es que esas nociones dependel del umbral.
Cabe resaltar que además de recibir el umbral, el constructor de MatrixTransformer también recibe el método inicial (601, 709 o 240) con el que se quiere hacer la transformación y el *path* de la imagen. Además, MatrixTransformer tiene dos setters setMethod que permite cambiar el método de transformación y setUmbral que permite cambiar el umbral que decidirá si el pixel se transforma a 0 o 1.

Finalmente, haremos una animación que ejemplifica una transformación de una imagen en otra. Para ello debemos analizar cada matching encontrado en la solución. Si en el matching entre A[i] y B[i] hay una división, entonces es razonable que el bloque correspondiente en A[i] se divida en subloques proporcionales a los tamaños de los bloques correspondientes en B[i]. Por ejemplo, si un bloque, de tamaño 14 mediante una división termina en tres bloques de tamaño 10, 20, y 5. Entonces dicho bloque será dividido en subloques de 4, 8, 2, cada uno de los cuales se irá transfor-

mando progresivamente en su correspondiente bloque en B[i]. Un razonamiento análogo occurre en el caso de una agrupación.

9. Animación

Implementar una animación para transformar imágenes. Deberá recibir como entrada dos imágenes y mostrar una animación que transforma la primera imagen en la segunda. Aunque internamente se trabaje con matrices de ceros y unos, esta animación deberá mostrar la transformación en colores.
Esta función deberá recibir adicionalmente un parámetro toma en cuenta el algoritmo a utilizar:

- Utiliza la subrutina implementada en la pregunta 6 (Voraz).
- Utiliza la subrutina implementada en la pregunta 7 (Prog. Dinámica).
- Utiliza la subrutina implementada en la pregunta 10 (Prog. Dinámica Mejorada).

Nosotros realizamos dos métodos para transformar las imágenes.

1. Utilizando los matches: Ya sea una agrupación o división, nosotros podemos ver al matching como dos filas. Supongamos que tenemos 3 bloques de 3, 4 y 2 bits que se agrupan en 1 bloque de 8 bits, al ver a estos tres primeros bloques como 1 solo lo que realizamos es lo siguiente: transformamos el primer bit del primer bloque, con el primer bit del segundo bloque (aumentando o disminuyendo el RGB de este bit). En caso uno de los bloques tenga más bits que el otro, este proceso se hará hasta que el bloque con menor bits termine.
2. Pixel por pixel: Este método se basa en hacer una transformación de los bits uno a uno. No utiliza los matches, simplemente se transforma el bit i, j de la primera matriz al bit i, j de la segunda matriz.

10. Transformación Programación Dinámica Mejorada

Analice, diseñe e implemente un algoritmo de programación dinámica con complejidad cúbica para el problema MIN-MATCHING. Su algoritmo deberá devolver la transformación de peso *promedio* mínimo. Deberá usar como subrutina un algoritmo que encuentre el peso promedio mínimo de un matching.

Tiempo de ejecución del algoritmo: $\mathcal{O}(pq^3)$.

Require: Dos matrices A y B de ceros y unos de tamaño p x q.
Ensure: Una transformación óptima entre A y B, y su peso.

MIN-TRANSFORMACION-MEJORADA(A, B)**:**

Sea A' la matriz donde cada fila tiene los bloques de cada fila de A, $\forall_{1 \leq i \leq p}$.
Sea B' la matriz donde cada fila tiene los bloques de cada fila de B, $\forall_{1 \leq i \leq p}$.
Sea *peso* la variable que va a guardar el peso del Min-Matching entre A y B.
Sea M una matriz donde cada fila i de M contiene el min-matching entre la fila i de A y B, $\forall_{1 \leq i \leq p}$. Es decir, cada fila i de M es un array de pairs que representan el min-matching entre la fila i de A y B.

$peso \leftarrow 0$

for i = 1 **to** p **do**
    Sea A'_i la i-ésima fila de A'
    Sea B'_i la i-ésima fila de B'
    $pair \leftarrow$ **Min-Matching-Mejorado**(A'$_i$, B'_i)
    $peso \leftarrow$ peso + $pair.second$
    $M[i] \leftarrow pair.first$
end for

Sea $respuesta$ un pair tal que:
$respuesta.first \leftarrow M$

18: $respuesta.second \leftarrow peso$
19:
20: **return** *respuesta*

- Min-Matching-Mejorado realiza una transformación mejorada entre dos arreglos de 1s y 0s.

Require: Dos arreglos A y B de ceros y unos, de tamaño p con n y m bloques, respectivamente.
Ensure: Un matching entre A y B, de peso mínimo, y su peso.

Min-Matching-Mejorado(A, B)**:**
1: Sea A' el arreglo con los bloques de A con n bloques.
2: Sea B' el arreglo con los bloques de B con m bloques
3: Sea X el vector de las conexiones que se van a realizar entre A' y B'.
4: Sea $Respuesta_i$ un $pair < vector, float >$ que contiene X y el peso.
5: Sea M una matriz declarada globalmente que contiene las *Respuestas*.
6: $sum \leftarrow 0$
7: $VarianzaA \leftarrow 0$
8: $VarianzaB \leftarrow 0$
9: **for** i = 1 **to** n **do**
10: Respuesta'.first.emplace_back($A'_1 \cup \ldots \cup A'_i$, B'_1)
11: $sum \leftarrow sum + A_i$
12: Respuesta'.second $\leftarrow sum$
13: M[i][1] = Respuesta'
14: **end for**
15:
16: $VarianzaA \leftarrow sum$
17: $sum \leftarrow 0$
18:
19: **for** j = 1 **to** m **do**
20: Respuesta'.first.emplace_back(A'_1, $B'_1 \cup \ldots \cup B'_j$)
21: $sum \leftarrow sum + B_j$
22: Respuesta'.second $\leftarrow sum$
23: M[1][j] = Respuesta'
24: **end for**
25:
26: $VarianzaB \leftarrow sum$

27: $Varianza \leftarrow \frac{VarianzaA}{VarianzaB}$
28:
29: $min_1 \leftarrow \infty$
30: $min_2 \leftarrow \infty$
31: $sum_1 \leftarrow 0$
32: $sum_2 \leftarrow 0$
33: $weight_1 \leftarrow 0$
34: $weight_2 \leftarrow 0$
35: $conexiones_1 \leftarrow \emptyset$
36: $conexiones_2 \leftarrow \emptyset$
37:
38: **for** i = 2 **to** n **do**
39: **for** j = 2 **to** m **do**
40: **for** k = j-1 **to** 1 **do**
41: $conexiones_1 \leftarrow \emptyset$
42: **for** a = k+1 **to** j **do**
43: $sum_1 \leftarrow B_a$
44: $conexiones_1 \leftarrow conexiones_1 \cup B_a$
45: **end for**
46: $conexiones_1 \leftarrow M[i-1][k].first \cup conexiones_1$
47: $weight_1 \leftarrow M[i-1][k].first + \frac{A_i}{sum_1}$
48: **if** $weight_1 < min_1$ **then**
49: $Respuesta_1.first \leftarrow weight_1$
50: $Respuesta_1.second \leftarrow conexiones_1$
51: **end if**
52: **end for**
53: **for** k = 1 **to** i-1 **do**
54: $conexiones_2 \leftarrow \emptyset$
55: **for** a = k+1 **to** i **do**
56: $sum_2 \leftarrow A_a$
57: $conexiones_2 \leftarrow conexiones_2 \cup A_a$
58: **end for**
59: $conexiones_2 \leftarrow M[k][j-1].first \cup conexiones_2$
60: $weight_2 \leftarrow M[k][j-1].second + \frac{sum_2}{B_j}$
61: **if** $weight_2 < min_2$ **then**
62: $Respuesta_2.first \leftarrow weight_2$

63: $Respuesta_2.second \leftarrow conexiones_2$
64: **end if**
65: **end for**
66:
67: **if** $min_1 < min_2$ **then**
68: $M[i][j].first \leftarrow Respuesta_1.first$
69: $M[i][j].second \leftarrow |Respuesta_1.second - Varianza|$
70: **else**
71: $M[i][j].first \leftarrow Respuesta_2.first$
72: $M[i][j].second \leftarrow |Respuesta_2.second - Varianza|$
73: **end if**
74: **end for**
75: **end for**
76: **return** $M[n][m]$

Análisis de Tiempo

Sabemos que el costo de **Min-Matching-Mejorado** es de $\mathcal{O}(m^2n)$, ya que hace exactamente lo mismo que **Min-Matching**, pero se le agrega un trabajo constante: inicialización de tres variables en las líneas 7, 8 y 27; actualización de estas variables en las líneas 16 y 26; y finalmente, una resta más en las líneas 69 y 72.

En $\mathcal{O}(m^2n)$, n representa el número de bloques de A y m es el número de bloques de B, donde A y B tienen tamaño q. Observe que si **Min-Matching-Mejorado** corre en $\mathcal{O}(m^2n)$, entonces también corre en $\mathcal{O}(q^3)$, ya que siempre el número de bloques será menor al número de bits en el arreglo. Entonces, como **Min-Matching-Mejorado** corre en $\mathcal{O}(q^3)$ y llamamos a la función **Min-Matching-Mejorado** p veces, podemos afirmar que el algoritmo **Min-Transformacion-Mejorada** corre en $\mathcal{O}(pq^3)$.

Implementación

Las implementaciones se encuentran en el siguiente repositorio de GitHub:

- `https://github.com/MenuMarino/Proyecto-ADA`