Proyecto del curso Análisis y Diseño de Algoritmos

1st Piero Morales Alcalde
Computer Science Student
Universidad de Ingeniería y Tecnología
Lima, Perú
piero.morales@utec.edu.pe

2nd André Segovia Melgarejo
Computer Science Student
Universidad de Ingeniería y Tecnología
Lima, Perú
andre.segovia@utec.edu.pe

3rd Osman Vilchez Aguirre
Computer Science Student
Universidad de Ingeniería y Tecnología
Lima, Perú
osman.vilchez@utec.edu.pe

I. Introduction

En este proyecto se va resolver el problema de MIN-MATCHING con el uso de algoritmos de tipo Greedy y con Programación dinámica.

II. Problema de Min-Matching

Dados dos vectores A y B de ceros y unos, se debe encontar un **matching de peso mínimo**. En cada uno de los vectores se pueden encontrar bloques. Un *bloque* es un subarreglo de unos. Cada bloque puede ser denotado por un par ordenado $[i, j]$, donde i es el índice inicial del bloque y j es el índice final del bloque. Por ejemplo, si A tiene m bloques, entonces se pueden ordenar dichos bloques de forma creciente por índice inicial.

Ahora, se quiere asociar los segmentos de A con los segmentos de B. Más formalmente, un *matching* entre A y B es un conjunto M de pares ordenados (i, j) que comple las siguientes condiciones.

1) Todo índice entre 1 y m aparece alguna vez en la primera coordenada de algún par ordenado en M. Todo índice entre 1 y n aparece alguna vez en la segunda coordenada de algún par ordenado en M.

2) Si $(i_1, j_2) \in M$, $i_1 < i_2$ y $j1 < j2$, entonces $(i_2, j_1) \notin M$.

3) Si (i_1, j_1), $(i_2, j_2) \in M$ con $i_1 < i_2$ y $j_1 < j_2$, entonces (i_1, j_2), $(i_2, j_1) \notin M$.

Es decir, un *matching* corresponde a una transformación entre bloques de A hacia los bloques de B, tal que algunos bloques de A son divididos y otros bloques son agrupados.

Formalmente, dado un *matching* entre dos vectores, A y un índice i, una i-división es un subconjunto $(i, j_1), (i, j_2), ..., (i, j_k)$ del matching original. Note que, debido a la definición de matching, $j_1, j_2, ..., j_k$ son índices consecutivos. Y dado un índice j, un j-agrupamiento es un subconjunto $(i_1, j), (i_2, j), ..., (i_l, j)$ del matching original. Note que, debido a la definición de matching, $i_1, i_2, ..., i_l$ son índices consecutivos.

Si $D = \{(i, j_1), (i, j_2), ..., (i, j_k)\}$ es una división, entonces el peso asociado a dicha división es el siguiente.

$$w(D) = \frac{|A_i|}{|B_{j_1}|+|B_{j_2}|+...+|B_{j_k}|}$$

Si $D = \{(i_1, j), (i_2, j), ..., (i_l, j)\}$ es una agrupación, entonces el peso asociado a dicha agrupación es la siguiente.

$$w(D) = \frac{|A_{i_1}|+|A_{i_2}|+...+|A_{i_l}|}{|B_j|}$$

El peso de un matching M, denotado por $w(M)$, es definido como la suma de los pesos de las agrupaciones y las divisiones en M, como se muestra a continuación.

$$w(M) = \sum_{D:D\ es\ una\ division\ o\ agrupacion\ en\ M} w(D)$$

III. Pregunta 1: Algoritmo voráz

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

Entrada del algoritmo: Dos arreglos A y B de ceros y unos de tamaño p, con n bloques y m bloques respectivamente (los valores de n y m no son recibidos como entrada).

Salida del algoritmo: Un matching entre A y B, no necesariamente óptimo, y su peso.

Tiempo de ejecución del algoritmo: $O(max\{m, n\})$.

Como solución voráz al problema de MIN-MATCHING primero se ha construido un algoritmo que convierte los dos vectores de unos y ceros iniciales en dos vectores de números enteros que se envían como argumentos al algoritmo principal.

El algoritmo para la conversión de vectores se muestra a continuación.

Algorithm 1 VECTOR-CONVERTER

Require: Un arreglo A de unos y ceros.
Ensure: Un arreglo A' con la suma de unos para cada bloque.

```
VECTOR-CONVERTER(A)
count = 0
values = [∅]
for i in A do
    if i = 1 then
        count = count + 1
    else
        if count ≠ 0 then
            values.add(count)
            count = 0
        end if
    end if
end for
if count ≠ 0 then
    values.add(count)
end if
return values
```

Luego de realizar la conversión de vectores de unos y ceros en vectores de número enteros, se puede pasar a realizar el matching entre los dos nuevos vectores. El algoritmo para este matching se detalla como MIN-MATCHING.

Por ejemplo, supongamos que tenemos los siguientes vectores como entrada inicial.

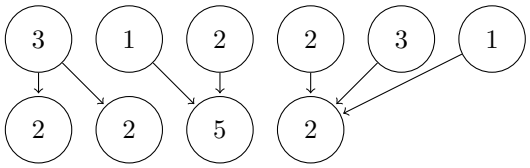
$$A = [0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0]$$
$$B = [0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0]$$

Cada uno de estos vectores se ejecutarían sobre el algoritmo CONVERT-VECTOR, el cual recibe como argumento un vector de unos y ceros. Luego de pasar ambos vectores sobre dicho algortimo, obtenemos como resultado los siguientes vectores de números enteros.

$$A = [3, 1, 2, 2, 3, 1]$$
$$B = [2, 2, 5, 2]$$

Estos dos vectores, luego van a ser enviados como argumentos al algortimo principal MIN-MATCHING que se va a encargar de hacer el matching entre estos dos vectores. Como el algortimo no necesariamente nos tiene que devolver el matching de peso mínimo, se puede hacer la siguiente elección voráz.

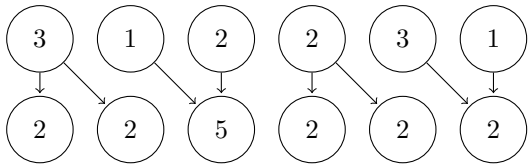
Elección voráz: Realizar divisones y agrupamientos de forma intercalada desde el inicio hasta el final de los vectores, como se observa a continuación.



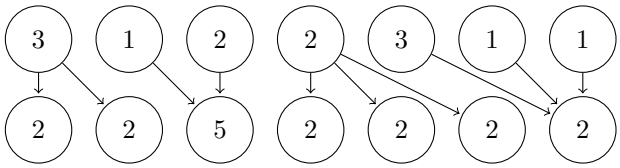
Para esta elección voraz se presentan las siguientes condiciones que hay que tener en cuenta.

- Si los dos vectores de entrada (A y B) son iguales,
  - Si el tamaño de los vectores es multiplo de 3.
  - Si el tamaño de los vectores es multiplo de $3 + 1$.
  - Si el tamaño de los vectores es multiplo de $3 + 2$.
- Si las dos cadenas de entrada (A y B) son diferentes.

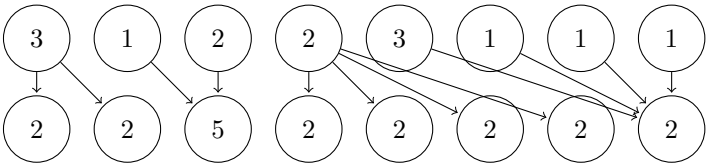
Cuando se tiene dos vectores cuyo tamaño son múltiplos de 3, se puede hacer divisiones y agrupamientos de bloques de manera intercalada sin ningún problema. De esta manera, siempre se va a obtener *i-divisiones* de la forma $(i, j_1), (i, j_2)$ y *j-agrupamientos* de la forma $(i_1, j), (i_2, j)$, como se muestra a continuación.



Para el caso cuando el tamaño de los vectores es un múltiplo de 3 más 1, al comienzo, se realizan las divisiones y agrupaciones de la misma manera, hasta que queden 4 elementos por asignar. En este punto, se hace una división entre el primer elemento de A que aún no ha sido asignado con todos los elementos sin asignar de B, menos el último. Luego, se hace una agrupación entre los elementos restantes de A con el último elemento de B. Esta forma de match se muestra a continuación.

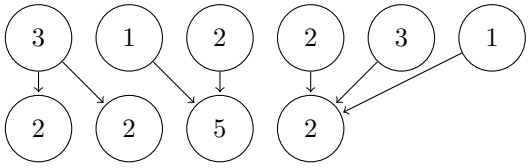


Por último, para el caso cuando el tamaño de los vectores es un múltiplo de 3 más 2, al comienzo, se realizan las divisiones y agrupaciones de la misma manera que el caso 1, hasta que queden 5 elementos por asignar. En este punto, se hace una división entre el primer elemento de A que aún no ha sido asignado con todos los elementos sin asignar de B, menos el último. Luego, se hace una agrupación entre los elementos restantes de A con el último elemento de B. Esta forma de match se muestra a continuación.



Ahora, para los casos en los que los vectores tienen diferentes tamaños, se debe hacer una división de ambos vectores hasta que tengan el tamaño del vector menor menos 1. Así, tendríamos dos vectores del mismo tamaño que se puede procesar con cualquiera de las maneras ya mencionadas. Y

para los nodos restantes, quedarían hacer una divisón si es que el tamaño original del vector A es menor que el tamaño original de B, y un agrupamiento en caso contrario, como se muestra a continuación.



El pseudocódigo de este algoritmo voraz para el problema se muestra en el algoritmo 2 MIN-MATCHING-VORAZ.

Algorithm 2 MIN-MATCHING-VORAZ

Require: Dos arreglos A y B con la suma de unos por bloque.
Ensure: Un matching entre A y B, y su peso.

```
MIN-MATCHING-VORAZ(A, B)
result = ∅
if A.length = B.length then
    if A.length mod 3 = 0 then
        result = MATCH-MULT-3(A, B)
    else if A.length mod 3 = 1 then
        result = MATCH-MULT-3-1(A, B)
    else
        result = MATCH-MULT-3-2(A, B)
    end if
else
    if A.length > B.length then
        A' = A[1 : B.length − 1]
        B' = B[1 : B.length − 1]
        if A'.length mod 3 = 0 then
            result = MATCH-MULT-3(A', B')
        else if A'.length mod 3 = 1 then
            result = MATCH-MULT-3-1(A', B')
        else
            result = MATCH-MULT-3-2(A', B')
        end if
        t = A[B.length : A.length]
        result.add(t, B[B.length])
    else
        A' = A[1 : A.length − 1]
        B' = B[1 : A.length − 1]
        if A'.length mod 3 = 0 then
            result = MATCH-MULT-3(A', B')
        else if A'.length mod 3 = 1 then
            result = MATCH-MULT-3-1(A', B')
        else
            result = MATCH-MULT-3-2(A', B')
        end if
        t = A[A.length : B.length]
        result.add(A[A.length], t)
    end if
end if
return result
```

IV. PREGUNTA 2: RECURRENCIA

Tenemos la siguiente recurrencia para el problema de MIN-MATCHING.

$$OPT(i,j) = \begin{cases} \frac{A_i}{\sum_{k=1}^{i} B_k} & \text{para todo } i = 1 \\ \frac{\sum_{k=1}^{i} A_k}{B_j} & \text{para todo } j = 1 \\ min(min_{l=1}^{j-1}\{OPT(i-1, & \text{otro caso} \\ j-l) + \frac{A_i}{\sum_{k=j-l+1}^{j} B_k}\}; & \\ min_{l=1}^{i-1}\{OPT(i-l, j-1) & \\ + \frac{\sum_{k=i-l+1}^{i} A_k}{B_j}\}) & \end{cases}$$

V. PREGUNTA 3: RECURSIVO

A. Recurrencia de complejidad

Tenemos la siguiente recurrencia de complejidad para el problema de MIN-MATCHING-RECURSIVO.

$$C(m,n) = \begin{cases} 2 * c_1 & \text{para todo } i = 2 \text{ ó } j = 2 \\ \sum_{n=1}^{n-1} C(m-1, n-h) + & \text{otro caso} \\ \sum_{n=1}^{m-1} C(m-h, n-1) + & \\ \Omega(1) & \end{cases}$$

B. Demostración de complejidad

Tenemos lo siguiente:

$C(m;n) = \sum_{n=1}^{n-1} C(m-1; n-h) +$

$\sum_{n=1}^{m-1} C(m-h; n-1) + \Omega(1) \geq C(m-1; n-1) +$

$C(m-1; n-1)$

$$\therefore C(m;n) \geq C(m-1;n-1) + C(m-1;n-1)$$

$$\boxed{C(m;n) \geq 2C(m-1;n-1)}$$

Hipótesis: $C(m;n) = \Omega(2^r)$, r es máximo
$\Rightarrow C(m;n) \geq C \cdot 2^r \leftrightarrow m \wedge n \neq 1$

Adicional: $C(2;2) = 2 \cdot K \geq 2^2 \cdot C$
$k = 2 \wedge C = 1$

$$\begin{aligned} C(2;2) &= \Omega\left(2^2\right) \\ &\vdots \\ C(m-1;n-1) &= \Omega\left(2^{r-1}\right) \\ C(m;n) &= \Omega(2^r) \end{aligned}$$

Por hipótesis inductiva, tenemos:

$C(m;n) \geq 2 \cdot C(m-1;n-1)$

$2^r \cdot K \geq 2 \cdot 2^{r-1} \cdot K_1$

$K \cdot 2 \wedge K_1 = 1$ **Se verifica**

$C(m;n) = \Omega\left(2^{max\{m,n\}}\right);\ m \wedge n \neq 1$

C. Algoritmo recursivo

Para resolver el problema de MIN-MATCHING se ha planteado el siguiente pseudocódigo.

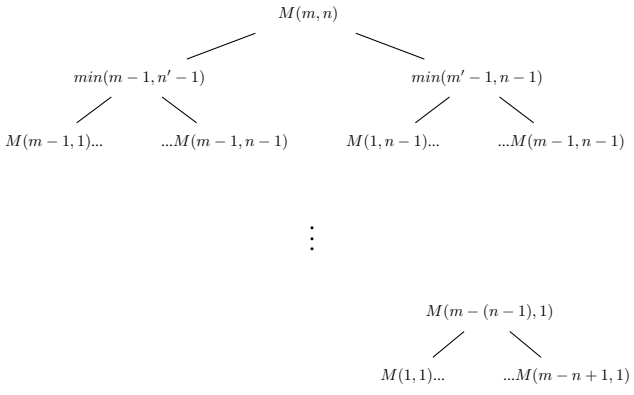
Algorithm 3 MIN-MATCHING-RECURSIVO

Require: Dos arreglos A y B con la suma de unos por bloque.
Ensure: Un matching entre A y B, y su peso.

```
MIN-MATCHING-RECURSIVO(A, B)
i = A.length
j = B.length
if i = 1 and j = 1 then
    return {(A[i], B[j])}
else if i = 1 or j = 1 then
    if i = 1 then
        match = (A[i], ∅)
        for a = 1 to j + 1 do
            match[2].add(B[a])
        end for
        return {match}
    else
        match = (∅, B[j])
        for a = 1 to i + 1 do
            match[1].add(A[a])
        end for
        return {match}
    end if
else
    pesos = ∅
    match = MIN-MATCHING-RECURSIVO(A[1 : i], B[1 : j])
    match.add(A[i], B[j])
    pesos.add(match)
    for a = 1 to i do
        matches = MIN-MATCHING-RECURSIVO(A[1 : a], B[1 : j])
        match = (∅, B[j])
        for b = a to i + 1 do
            match[1].add(A[b])
        end for
        matches.add(match)
        pesos.add(matches)
    end for
    for a = 2 to j do
        matches = MIN-MATCHING-RECURSIVO(A[1 : i], B[1 : a])
        match = (A[i], ∅)
        for b = a to j + 1 do
            match[2].add(B[b])
        end for
        matches.add(match)
        pesos.add(matches)
    end for
    return min(pesos)
end if
```

VI. PREGUNTA 4: MEMORIZADO

A. Arbol de análisis de costo



B. Recurrencia de complejidad

Tenemos la siguiente recurrencia para el problema de MIN-MATCHING-MEMORIZADO.

$$C(m,n) = \begin{cases} c_1 * n & \text{para todo } i = 1 \\ c_1 * m & \text{para todo } j = 1 \\ C(m-1, n-1) + k(n-2) + d & \text{otro caso} \end{cases}$$

C. Demostración de complejidad

Suponiendo que $m > n$:

$$\begin{aligned} &C(m,n) = C(m-1,n-1) + k(m-2) + d \\ &= C(m-2,n-2) + k(m-2) + k(m-3) + 2d \\ &= C(m-2,n-2) + 2km - k(2+3) + 2d \\ &= C(m-3,n-3) + 3km - k(2+3+4) + 3d \\ &\vdots \\ &= C(m-(n-1), n-(n-1)) + (n-1)km - \\ &k(2+3+4...+n) + (n-1)d \\ &= C(m-(n-1),1) + (n-1)km - k(\tfrac{n*(n+1)}{2} - 1) + (n-1)d \\ &= O(m) + (n-1)km + O(n^2) + O(n) \end{aligned}$$

Se verifica que $O(m)$, $O(n)$ y $O(n^2)$ son $O(m*n)$
Entonces:
Faltaría demostrar que $(n-1)km = O(m*n)$
$k*nm - k*m \leq c*mn$
$k - \frac{k}{n} \leq c$
Con c = 1 n_0=1; $n \geq n_0 > 0$
$0 \leq c$
$\Rightarrow (n-1)km = O(n*m)$

Conluimos: $C(m,n) = O(n*m)$

VII. PREGUNTA 5: PROGRAMACIÓN DINÁMICA

El algoritmo en Programación Dinámica esta basado en el Memorizado principalmente, al ser este muy largo optamos por no colocarlo aquí, sin embargo se puede visualizar en el **siguiente enlace**. En este algoritmo se utilizan variables que actúan tanto como stacks como contadores, entre estas están:

- **min**: Aquí se almacena el match mínimo, este va actualizándose conforme se vayan descubriendo nuevos matches
- **mem**: Aquí se almacena la matriz de memoria, esta crece conforme se van descubriendo nuevos matches
- **actual**: Esta variable actúa como stack en donde se van formando los distintos matches, cuando esta completo, si es que su valor total es menor que el mínimo actual, esta se vuelve el mínimo actual
- **counters**: Aquí almacenamos un stack de contadores que se encarga de actualizar en tiempo real las combinaciones de configuraciones posibles dentro de un match
- **possible_values**: Aquí almacenamos un stack que va sincronizado con counters y que se encarga de ir guardando los matches que sirven para buscar el mínimo en su nivel que a su vez ayuden a formar la memoria
- **i, j**: Aquí se almacenan las posiciones en el primer y segundo vector

Algorithm 4 MIN-MATCHING-MEMOIZADO

Require: Dos arreglos A y B con la suma de unos por bloque.
Ensure: Un matching entre A y B, y su peso.

```
MIN-MATCHING-MEMOIZADO(A, B)
i = A.length
j = B.length
if i = 1 and j = 1 then
    return {(A[i], B[j])}
else if i = 1 or j = 1 then
    if i = 1 then
        match = (A[i], ∅)
        for a = 1 to j + 1 do
            match[2].add(B[a])
        end for
        return {match}
    else
        match = (∅, B[j])
        for a = 1 to i + 1 do
            match[1].add(A[a])
        end for
        return {match}
    end if
else
    if memoria.get(i) then
        if memoria[i].get(j) then
            return memoria[i][j]
        end if
    end if
    pesos = ∅
    match = MIN-MATCHING-MEMORIZADO(A[1 : i], B[1 : j])
    match.add(A[i], B[j])
    pesos.add(match)
    for a = 2 to i do
        matches = MIN-MATCHING-MEMORIZADO(A[1 : a], B[1 : j])
        match = (∅, B[j])
        for b = a to i + 1 do
            match[1].add(A[b])
        end for
        matches.add(match)
        pesos.add(matches)
    end for
    for a = 2 to j do
        matches = MIN-MATCHING-MEMORIZADO(A[1 : i], B[1 : a])
        match = (A[i], ∅)
        for b = a to j + 1 do
            match[2].add(B[b])
        end for
        matches.add(match)
        pesos.add(matches)
    end for
    minp = min(pesos)
    if memoria.get(i) then
        memoria[i][j] = minp
    else
        memoria[i] = dic()
        memoria[i][j] = minp
    end if
    return memoria[i][j]
end if
```

VIII. TRANSFORMACIÓN DE IMÁGENES

Dadas dos matrices $A[1...p, 1...q]$, $B[1...p, 1...q]$ de ceros y unos, una *transformación* de A en B es un conjunto $M = \{M_1, M_2, ..., M_p\}$, donde cada M_i es un matching entre los vectores $A[i]$ y $B[i]$. El peso de M, denotado por $w(M)$ es igual a la suma de pesos de cada M_i es decir, de acuerdo a lo siguiente.

$$w(M) = \sum_{i=1}^{n} w\left(M_i\right)$$

A. Problema Min-Transformacion

Dadas dos matrices A y B de ceros y unos, encontrar una transformación entre A y B de peso mínimo.

Es claro que para resolver el problema MIN-TRANSFORMACION de manera óptima basta invocar varias veces a alguna de las subrutinas implementadas en la sección anterior.

IX. PREGUNTA 6: TRANSFORMACIÓN VORÁZ

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

Entrada del algoritmo: Dos matrices A y B de ceros y unos de tamaño $p \times q$.

Salida del algoritmo: Una transformación entre A y B, no necesariamente óptima, y su peso.

Tiempo de ejecución del algoritmo: $O(pq)$

Para el planteamiento de la solución voráz a este problema, primero se ha realizado una llamada iterativa al Algoritmo 1 para realizar la conversión de una matriz de unos y ceros a una matriz de pesos de los bloques (números enteros). Entonces, a partir de esto se puede construir este nuevo algoritmo.

Algorithm 5 MATRIX-CONVERTER

Require: Una matriz A de unos y ceros.
Ensure: Una matriz A' con la suma de unos para cada bloque.

```
MATRIX-CONVERTER(A)
resultado = ∅
for row in A do
    resultado = resultado ∪ VECTOR-CONVERTER(row)
end for
return resultado
```

Ahora, se ha planteado como solución para el problema de *Min-Transformación* llamar de forma iterativa al Algoritmo 2 para cada fila de las matrices. Como dice el planteamiento del problema, se debe de hacer un MATCH entre las filas de ambas matrices, y el índice de la fila sea el mismo en ambas. A partir de esto, se ha planteado el siguiente algoritmo.

Algorithm 6 MIN-TRANSFORMACION-VORAZ

Require: Dos matrices A y B con la suma de unos por bloque.
Ensure: Una transformación entre A y B, y su peso.

```
MIN-TRANSFORMACION-VORAZ(A, B)
resultado = ∅
peso = 0
for i = 0 to A.rows do
    resultado = resultado ∪ MIN-MATCHING-VORAZ(A[i], B[i])
end for
for i = 0 to resultado.size do
    peso = peso + SUM(resultado[i])
end for
return [resultado, peso]
```

Para poder calcular la suma de los *match* de cada fila, se ha creado una función, la cual se muestra a continuación.

Algorithm 7 SUM

Require: Un *match* entre dos arreglos.
Ensure: El peso del *match* inicial.

```
SUM(match)
if match.length = 0 then
    return 0
end if
resultado = 0
for i in match do
    resultado = resultado + peso(i)
end for
return resultado
```

X. PREGUNTA 7: TRANSFORMACIÓN PROG. DINÁMICA

Para la transformacion de Programacion Dinamica se aplico la misma estrategis vista previamente simplemente reemplazando la funcion que haya los matchings.

Algorithm 8 MIN-TRANSFORMACION-DINAMICO

Require: Dos matrices A y B con la suma de unos por bloque.
Ensure: Una transformación entre A y B, y su peso.

```
MIN-TRANSFORMACION-DINAMICO(A, B)
resultado = ∅
peso = 0
for i = 0 to A.rows do
    resultado = resultado ∪ MIN-MATCHING-DINAMICO(A[i], B[i])
end for
for i = 0 to resultado.size do
    peso = peso + SUM(resultado[i])
end for
return [resultado, peso]
```

XI. PREGUNTA 8: LECTURA DE IMÁGENES

La motivación de hacer transformación de una matriz hacia otra es poder transformar una imagen en otra mediante una curva suave. Una manera de hacerlo es codificar cada pixel como un 0 o un 1. Para ello, puede tomar cada pixel en la escala RGB (r, g, b) y transformarlo a una escala de grises, para posteriormente escoger los que están más cerca de ser pixeles blancos (0) y los que están más cerca de ser negros (1). Existen muchos métodos para hacer esta transformación, dependiendo de los coeficientes escogidos.

Las fórmulas correspondientes para realizar esta conversión, puede ser con Rec. 601 Luma, la cual puede ser calculada con cualquiera de las siguientes fórmulas.

$$Y'_{601} = 0.299R' + 0.587G' + 0.114B' \quad (1)$$

$$Y'_{709} = 0.2126R' + 0.7152G' + 0.0722B' \quad (2)$$

$$Y'_{240} = 0.212R' + 0.701G' + 0.087B' \quad (3)$$

Aplicando estas fórmulas, se consigue construir una matriz nueva en escala de grises. Es decir, se ha realizado una transformación de R^3 a R^2. El proceso de transformación de esta matriz, sigue la siguiente forma.

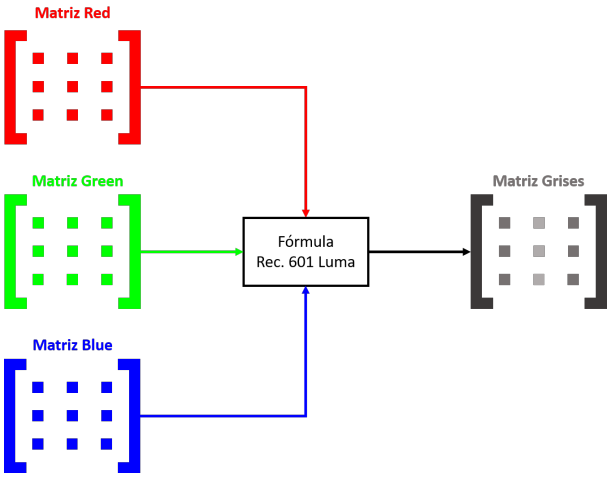


Fig. 1. Transformación de matrices RGB en matriz de escala de grises

Finalmente, para convertirlo a una matriz de unos y ceros, se define un umbral, para que se defina para cada casilla de la matriz de escala de grises, si esta se va a convertir en 1 o 0, como se muestra a continuación.

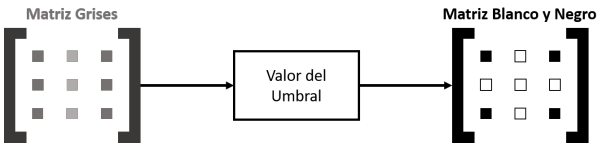


Fig. 2. Transformación de matriz de escala de grises en matriz de 1's y 0's

XII. PREGUNTA 9: ANIMACIÓN

Se va a implementar una animación que va a mostrar el proceso de transformación de una imagen en otra usando los algoritmos de transformación VORÁZ, PROG. DINÁMICA y PROG. DINÁMICA MEJORADA.

Para construir la animación, va a ser necesario crear imágenes intermedias que van a mostrar un pequeño cambio cuando se va a ir pasando de una en una. De esta manera, cuando se reproduzcan de manera rápida, se va a poder visualizar en forma de animación. Para la construcción de estas imágenes intermedias, se van a tener en cuenta los siguientes casos.

A. Agrupación

Cuando se tiene una agrupación en una transformación, lo que se tiene que hacer es calcular la cantidad de píxeles que se van a ir moviendo por cada imagen intermedia.

Para este caso, se debe realizar el cálculo de cuantas imágenes intermedias se van a querer generar, así como también, el tamaño de los bloques de unos de ambas matrices para saber qué cantidad de píxeles se van a convertir en la imagen resultante. Este proceso se puede ver en la siguiente imagen.

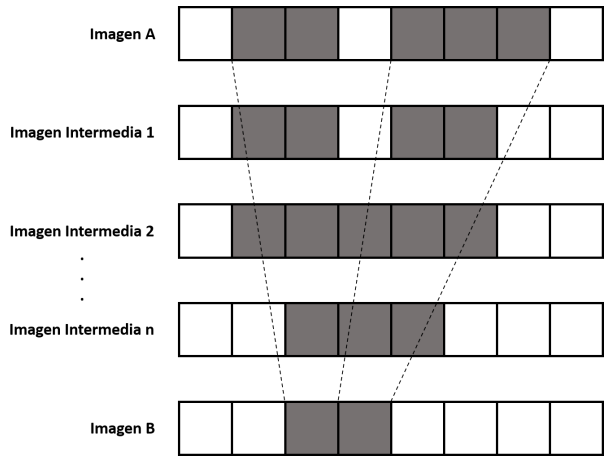


Fig. 3. Creación de imágenes intermedias para una agrupación

B. División

Cuando se tiene una división en una transformación, lo que se tiene que hacer es calcular la cantidad de píxeles que se van a ir moviendo por cada imagen intermedia.

Para este caso, se debe realizar el cálculo de cuantas imágenes intermedias se van a querer generar, así como también, el tamaño de los bloques de unos de ambas matrices para saber qué cantidad de píxeles se van a convertir en la imagen resultante. Este proceso se puede ver en la siguiente imagen.

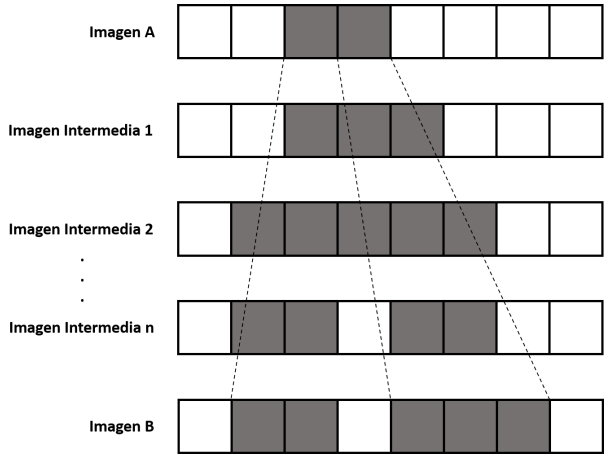


Fig. 4. Creación de imágenes intermedias para una división

XIII. Pregunta 10: Dinámica Mejorada

La mayor diferencia respecto a el dinámico normal fue la variación de nuestra función de peso la cual ahora recibía también el valor u como parámetro el cual se hallaria dentro de nuestra función dinámica mejorada

Anexos

Link repositorio GitHub: Proyecto ADA

Algorithm 9 MIN-TRANSFORMACION-MEJORADO

Require: Dos matrices A y B con la suma de unos por bloque.
Ensure: Una transformación entre A y B, y su peso.

MIN-TRANSFORMACION-DINAMICO-MEJORADO(A, B)
$resultado = \emptyset$
$peso = 0$
for $i = 0$ to $A.rows$ **do**
  $resultado = resultado \cup$ MIN-MATCHING-MEJORADO($A[i], B[i]$)
end for
for $i = 0$ to $resultado.size$ **do**
  $peso = peso +$ SUM($resultado[i]$)
end for
return $[resultado, peso]$

Algorithm 10 PESO

Require: Una configuración dentro de un match y el valor u
Ensure: El valor de la configuracion de el match

PESO($conf, u$)
if isinstance($conf$[0], int) **and** isinstance($conf$[1], int) **then**
  return abs($conf[0]/conf[1] - u$)
else if isinstance($conf$[0], int) **then**
  $temp$ = $conf$[0]
  sum = 0
  for i = 0 to len($conf$[1]) **do**
    sum = sum + $conf$[1][i]
  end for
  return abs($temp/sum - u$)
else
  $temp$ = $conf$[1]
  sum = 0
  for i = 0 to len($conf$[0]) **do**
    sum = sum + $conf$[1][i]
  end for
  return abs($sum/temp - u$)
end if