Proyecto ADA, primer entregable

Jean Miraval y Mateo Noel

July 25, 2020

1 Introducción

Para poder resolver el problema de Min-Matching con el uso de estos algoritmos primero decidimos convertir las cadenas de 0's y 1's en vectores de enteros.

Por ejemplo:
Una cadena A=[00111001], será representada como un vector A = {3,1}.

Esto es posible gracias a una función getBlocks:

Recibe:

- Vector A en binario (bool)

Devuelve:

- Vector A en pesos (int)

```
getBlocks(A)
1: vector <int> blocks
2: for i = 0 TO A.size()
3:    if A[i]
4:       while A[i]
5:          newBlock++
6:          i++
7:       blocks.push_back(newBlock);
8:    return blocks
```

-

Nota: El código igual recibe como input el tamaño de las cadenas, p, y estas mismas. Solo que las transformará a esta forma de representación.

2 Secuencias

Pregunta 1 (Voraz)

Para el diseño de un algoritmo voraz, se partió desde la intención de elegir las conexiones analizando únicamente la situación actual, es decir en qué bloque de

A o B estamos iterando. El objetivo del algoritmo es hallar el matching correcto entre los dos arrays que tenga el peso mínimo. Como se puede ver desde las fórmulas dadas, el peso de un Matching depende de una suma de pesos de diferentes conexiones. Entonces la estrategia voraz analizada deberá de buscar la manera de formar conexiones con el menor peso posible.

Estrategia: Armar conexiones minimizando el peso de estas.

Observando las fórmulas, podemos concluir que para minimizar el peso de estas, deberíamos procurar que la suma de los pesos de los bloques de B debe de ser mayor que las de A en cada conexión, para así minimizar el peso que resultará de la división. En base a esto, planteamos nuestra estrategia específica, la cual se basa en formar conexiones buscando dentro de lo posible que la suma de los bloques de A sea menos a la suma de los bloques de B.

Estrategia: Armar conexiones donde la suma de bloques de B sea mayor que las de A.

A partir de este análisis planteamos un pseudocódigo basado en analizar, a medida que avanzamos por una de las listas, si en todo momento la suma de los pesos de A y B permiten seguir haciendo conexiones de tal manera que cumpla con la estrategia. En caso contrario, se aplicaría un Reset Connection, que terminaría esa conexión e iniciaría una nueva a partir de los siguientes bloques. Se añadió las condiciones de combined y divided para evitar interferir con la condición del problema, donde un bloque de A no puede ser agrupado y dividido a la vez.

Recibe:

- Vectores A y B

Devuelve:

- Peso Min-Matching tipo Float

MIN-MATCHING-VORAZ(A, B)	*cost*	*times*
1: Divided=0	.	.
2: Combined=0	.	.
3: Peso=0	.	.
4: BCurrentw = w(B1)	.	.
5: ACurrentw = w(A1)	.	.
6: Conectar(1,1)	.	.
7: BCurrent = 1	.	.
8: ACurrent = 1	.	.
9: **for** $i = 2$ TO max(sizeA,sizeB)		
10: **if** BCurrent < sizeB and ACurrent < sizeA		
11: **if** ACurrentW < BCurrentW		
12: **if** ACurrent < sizeA-1		

```
13:             if Divided or Acurrentw + w(Ai) >= Bcurrentw
14:               RESET-CONECTION(...)
15:             else
16:               conectar(i,BCurrent)
17:               ACurrentw = ACurrentw + w(Ai)
18:               ACurrent = i
19:               combined = 1
20:               divided = 0
21:           else
22:             RESET-CONECTION(...)
23:         else
24:           if ACurrent < sizeA-1
25:             if if combined or BCurrent >= sizeB-1
26:               RESET-CONECTION(...)
27:             else
28:               BCurrent++
29:               i–
30:               conectar(A[ACurrent], B[BCurrent])
31:               divided =1
32:               combined=0
33:           else
34:             RESET-CONECTION(...)
35:         else if BCurrent ==sizeB
36:           ACurrent++
37:           conectar(A[ACurrent], B[BCurrent])
38:         else
39:           BCurrent++;
40:           conectar(A[ACurrent], B[BCurrent]);
41:           if BCurrent == sizeB-1
42:             break
43: peso += ACurrentw/BCurrentw
44: return peso
```

```
RESET-CONECTION(...)
1: BCurrent++
2: ACurrent++
3: peso += ACurrentw / BCurrentw
4: conectar(ACurrent, BCurrent)
5: ACurrentw = w(ACurrent)
6: BCurrentw = w(BCurrent)
```

Análisis de Tiempo

Como podemos observar en el código, la mayoría de las instrucciones dependen

de un costo constante y se repetiría las veces que se ejecute la línea del bucle. Este bucle itera el máximo entre el array de bloques de A de tamaño m y array de bloques de B de tamaño n. Todo el código dependerá del max(m,n)

$$T(m,n) = O(max\{m,n\})$$

Pregunta 2 (Recurrencia)

Para plantear el problema con recursividad, y con los conocimientos que se obtuvo al realizar el primer problema, nos dimos cuenta de que el solo añadir un bloque diferente, podría ocasionar en ciertos casos la reestructuración de todas las conexiones realizadas previamente. Por este motivo, no era posible realizar una recurrencia donde el resultado de OPT(i,j) esté relacionado directamente con una respuesta anterior. Por lo tanto, para el diseño de la recurrencia, se tomó en cuenta el uso de la recurrencia únicamente para mapear todas las combinaciones posibles del problema partiendo desde los casos iniciales, para posteriormente comparar cada una de ellas y elegir la más óptima, es decir para el problema. Por razones obvias, esta recurrencia es muy costosa en cuanto a recursos y tiempo.

Donde m = A.size y n = B.size

$$OPT(i,j) = \begin{cases} A_i / \sum_{x=j}^{n-1} B_x & i = m-1 \\ \\ \sum_{x=i}^{m-1} A_x / B_j & j = n-1 \\ \\ min(& \\ min_{w=i}^{m-2}(\sum_{x=i}^{w} A_x/B_j + OPT(w+1, j+1)) & \\ min_{w=j+1}^{n-2}(A_i / \sum_{x=j}^{w} B_x + OPT(i+1, w+1)) & \\ ) & \text{caso contrário} \end{cases}$$

Pregunta 3 (Recursivo)

Utilizando la recursividad planteada en la pregunta anterior, se pasó a ejecutar un algoritmo recursivo, donde utilizamos la recurrencia planteada para elegir el caso más óptimo para cada combinación de bloques de longitud m y n posibles con los inputs dados.

Recibe:

- a = donde iniciará A
- b = donde iniciará B
- vector A y B

Devuelve:

- Peso Min-Matching tipo Float

MIN-MATCHING-RECURSIVO(A, B, a, b)	*cost*	*times*
1: **if** a = m-1	.	.
2: return $A_a / \sum_{x=b}^{n-1} B_x$	.	.
3: **if** b = n-1	.	.
4: return $\sum_{x=a}^{m} A_x / B_b$	.	.
5: vector<int> posibles	.	.
6: **for** $i = a$ TO $m - 2$	.	.
7: p = $\sum_{x=a}^{i} A_x / B_b +$ Min-Match-Rec$(A, B, i + 1, b + 1)$	.	.
8: posibles.push_back(p)	.	.
9: **for** $i = b$ TO $n - 2$	.	.
10: p = $A_a / \sum_{x=b}^{i} B_x +$ Min-Match-Rec$(A, B, a + 1, i + 1)$	.	.
11: posibles.push_back(p)	.	.
12: return min(posibles)	.	.

Análisis de Tiempo
Por cada iteración se resolverán todos los subproblemas, entonces:

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

Pregunta 4 (Memoizado)

Partiendo desde el código anterior, se le añadirá únicamente el cambio de consulta de memoria del algoritmo, implementada a modo de una matriz. Antes de entrar a cálculos de la recursividad compleja, se consultará si el resultado ya ha sido calculado previamente. Además, en caso se ejecute con una combinación de parámetros que aún no ha sido calculada como solución, se almacenará esta solución a la matriz de memoria. De esta forma, luego de las ejecuciones iniciales, cualquier ejecución del algoritmo con una combinación de parámetros ya calculada, tendrá un tiempo de ejecución constante.

Recibe:

- a y b, índice de inicio dentro del subproblema en A y B
- Vector A y B
- M[m][n] donde se guardarán subproblemas

Devuelve:

- Peso Min-Matching tipo Float

MIN-MATCHING-MEMOIZADO(A, B, a, b, M) *cost* *times*

1: **if** M[a][b] . .
2: return M[a][b] . .
3: **if** a = m-1 . .
4: M[a][b] = $A_a / \sum_{x=b}^{n-1} B_x$ . .
5: return M[a][b] . .
6: **if** b = n-1 . .
7: M[a][b] = $\sum_{x=a}^{m} A_x / B_b$ . .
8: return M[a][b] . .
9: vector<int> posibles . .
10: **for** $i = a$ TO $m - 2$ . .
11: p = $\sum_{x=a}^{i} A_x / B_b$+
Min-Match-Mem$(A, B, i{+}1, b{+}1, M)$ . .
12: posibles.push_back(p) . .
13: **for** $i = b$ TO $n - 2$ . .
14: p = $A_a / \sum_{x=b}^{i} B_x$+
Min-Match-Mem$(A, B, a{+}1, i{+}1, M)$ . .
15: posibles.push_back(p) . .
16: M[a][b] = min(posibles)
17: return M[a][b] . .

Análisis de Tiempo
A medida que se va ejecutando la función vamos guardando valores en una matriz[m][n] y no tendrémos que ejecutar la función cada vez que se encuentre el mismo problema. Se ejecuta solo una iteración del Min-Matching-Memoizado a medida que se llena cada valor de la matriz, $T(m, n) <= c * (m * n)$ por ende:

$T(m, n) = O(mn)$

Pregunta 5 (Dinámico)

Es la aplicación de un algoritmo dinámico sobre la recurción previa

Recibe:

- Vector A y B

Devuelve:

- Peso Min-Matching tipo Float

MIN-MATCHING-DINÁMICO(A, B) *cost* *times*
1: M[m][n] . .

```
2:  for b = 0 TO n                                  .       .
3:     for i = b TO n                               .       .
4:        sumB = sumB+B[i]                          .       .
5:     M[m-1][b] = A[m-1] / sumB                    .       .
6:  for a = 0 TO m                                  .       .
7:     for i = a TO m                               .       .
8:        sumA = sumA+A[i]                          .       .
9:     M[a][n-1] = sumA / B[n-1]                    .       .
10: for a = m − 2 TO 0                              .       .
11:    for b = n − 1 TO 0                           .       .
12:       vector<float> posibles
13:       for i = a TO m − 2
14:          for j = a TO i
15:             sumA = sumA + A[j]
          posibles.push_back( sumA / B[b] + M[i+1][b+1]);
16:       for i = b TO n − 2
17:          for j = b TO i
18:             sumB = sumB + B[j]
          posibles.push_back(A[a] / sumB + M[a+1][i+1]);
19:    M[a][b] = min(posibles)
20: return M[0][0];
```

Análisis de Tiempo
Funciona similarmente al algoritmo memoizado, solo que este va creando los subproblemas mientras se va ejecutando, empezando por casos bases, subiendo hasta la respuesta. El tiempo de ejecución y espaciado sería el mismo del memoizado al usar el mismo concepto del algoritmo y espacio de memoria. $T(m, n) <= c * (m * n)$ por ende:

$$T(m, n) = O(mn)$$

3 Transformación de Imagenes

Pregunta 6 (Trasformación Voraz)

Anteriormente realizamos lo que sería el matching voraz, ahora necesitamos aplicarlo para hacer una transformación entre dos matrices. Para esto se ha iterado cada fila de las matrízes y se ha echo un min_matching voraz entre fila i de estas. El peso de esta transformación solo va a ser la suma de todos los matchings entre las matrices.
Ahora la función Min-Matching-Voraz también retorna un vector con los respectivos matchings para que puedan ser identificados.
p y q representan la cantidad de filas y columanas de la matriz booleana.

Recibe:

- Vector A y B

Devuelve:

- Peso Min-transformación tipo Float
- Un matching por cada fila de la matriz

TRANSFORMACION-VORAZ(A, B)

```
1:  for i = 0 to p
2:      vector<bool> A_i
3:      vector<bool> B_i
4:      for j = 0 to q
5:          A_i.push_back(A[i][j]);
6:          B_i.push_back(B[i][j]);
7:      total = Min-Matching-voraz(getBlocks(A_i),getBlocks(B_i)) + total;
8:      A_i.clear();
9:      B_i.clear();
10:     m_Match.push_back(vec_Match);
11:     return total
```

-

En la línea 10 metemos el match entre filas(vec_Match) a un vector de vectores que contará con todos los matches realizados(m_Match).

Análisis de Tiempo
Cómo realizamos la función Min-Matching-Voraz p veces el costo será:
$T() = O(pq)$

Pregunta 7 (Transformación Dinámica)

Igual que en la pregunta 6, en este caso esteríamos aplicando el min-matching dinámico sobre dos matrices para lograr la transformación.

Recibe:

- Matriz A y B

Devuelve:

- Peso Min-transformación tipo Float
- Un matching por cada fila de la matriz

Igual al Min_Matching_Voraz pero existe un cambio en la línea 7

TRANSFORMACIÓN-DINÁMICA(A, B)

7: total = Min-Matching-Dinámico(getBlocks(A_i),getBlocks(B_i)) + total;

Análisis de Tiempo
Como estamos realizando min-matching dinámico (el cual equivale a O(pq)) q veces el tiempo total será de:
$T() = O(pq^2)$

Pregunta 8 (Lectura de imágenes)

El siguiente paso consistió en leer imágenes como inputs, sobre las cuales se implementarán las funciones creadas en el ejercicio 6 y 7. Está dividido en tres pasos por imágen:

1. Decodificar los valores RGB
2. Guardar los datos de la imagen
3. Decodificar la data para obtener una matriz en binario que la represente

Decodificar los valores RGB

Utilizaremos una función que leerá el archivo que deseemos para guardar los tres valores numéricos RGB en un archivo de imagen .ppm.

```
INTERPRETER(image)
1: Inter = read(Image)
2: archivo = "img.ppm"
3: for i in RGB
4:     archivo.write(i);
```

-

Guardar los datos de la imagen.

Leemos la imagen como texto, y almacenamos todo en un vector. Recibe de input el nombre del archivo de imagen. Cada número que se lea, se almacenará en un vector de enteros.

```
READIMAGE(image)
1: image >> info #guardamos información de formato
2: archivo = "img.ppm"
3: while !Image.eof()
4:     image >> data;
5:     Parsed.push_back(data); #metemos los inputs en un vector
```

-

Decodificar la data para obtener una matriz en binario que la represente.

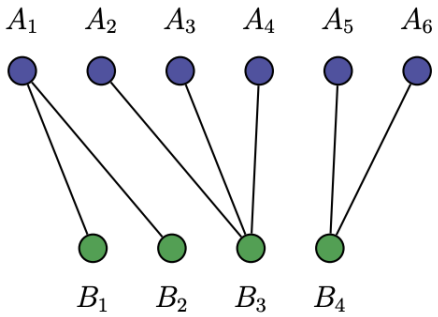
```
DECODEMAIN(parsed, A)
1: for c in parsed
2:     temp = decode(c) #convierte cada número a binario
3:     Matriz.append(temp)
```

Pregunta 9 (Animación)

Para realizar la animación tenemos que entender como se leerá el matching. La función Transformación nos devolverá 2 matrizes, uno que contienen vectores de pares del match y otro que contiene cuantos elementos tiene el match. Los cuales son llamados vec_match y vec_size, respectivamente.

Ejemplo:

- vec_match = $\{a_1, b_1\}, \{a_1, b_2\}, \{a_2, b_3\}, \{a_3, b_3\}, \{a_4, b_3\}, \{a_5, b_4\}, \{a_6, b_4\}$
- vec_size = $\{2, a\}, \{3, b\}, \{2, b\}$



vec_size[1] indica que tenemos 2 matches que salen de a.
vec_size[2] indica que tenemos 3 matches que salen de b.

Como ya tenemos el match de cada fila de las matrices, podemos empezar con la animación.

Matriz intermedia
Generaremos una matriz intermedia entre A y B que se generará gracias al matching. n. 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 ser irá transformando progresivamente en su correspondiente bloque en B[i].

Aplicación

El paso final fue implementar una animación de transformación de imágenes, utilizando como parte inicial la captura de inputs diseñada en el ejercicio 8, y permitiendo el ingreso de un input que seleccione qué método de transformación utilizar. El resultado de la función implementada será transformado a una matriz binaria que representa la imagen resultado. Estas imágenes serán calculadas y guardadas dentro de un bucle, que permitirán su visualización.

1. Convertir el output de las transformaciones a binario.

   Esta función recibe como parámetro la transformación del output de las funciones implementadas anteriormente, y las interpreta hasta convertirla en una matriz binaria Answer. Para esto, usaremos una subrutina llamada Transformar, que recibe un matching de una de las filas de las imágenes y las interpreta y transforma a binario.

   MatchtoBin(*image*)
   1: **for** for i in m_Match:
   2: Answer[i] = Traducir(m_Match, A[i], B[i])

   -
   En la función traducir es donde se empieza a crear la matriz intermedia. (Generándoce fila por fila de ambas matrices)

   Esta función lo que hace es agarrar los dos vectores A[i] y B[i]. Para generar el vector Mi[i] de la matriz intermedia.

   Por cada matching:
   011001000111 ———— 0001110000

   Realiza la operación: $\lceil x/y * n \rceil$
   "cantidad de matches que tenga el bloque único" veces

   Donde:
   x = peso de bloque único
   y = peso de bloques al que se conecta el bloque único
   n = peso de cada bloque separado

   Y genera nuevo segmento de vector:

   010001000110

   Cada nuevo peso de los bloques se almacenan en el vector que tenía los bloques divididos.

2. Guardar la imagen como un archivo ppm insertando valores RGB.

   Posteriormente , a partir de la matriz Answer resultante de la función

MatchtoBin, la podemos guardar como un texto de valores RGB y un formato .ppm utilizando la función WriteImage.

WriteImage($Answer, archivo$)

```
1: archivo.write(info);
2: for i in answer:
3:     temp = encode(i) #transforma una fila de binario a decimal.
4:     archivo.write(temp);
```

-

Primera fila: Se aprecia la animación realizada con la transformación voraz.
Segunda fila: Animación con transformación dinámica.

Pregunta 10 (Transformación Dinámica Mejorada)

Anteriormente con los algoritmos anteriores se buscaba maximizar B. Ahora con el uso de un nuevo algoritmo se podrá notar una curva más suave al momento de ejecutar la animación pporque se está buscando un equilibrio entre las dos imágenes

Recibe:

- Matriz A y B

Devuelve:

- Peso Promedio-transformación tipo Float
- Un matching por cada fila de la matriz

Aclaración

Transformación-Dinámica-Promedio(A, B)

Análisis de Tiempo
Misma complejidad de tiempo que la otra tranformación dinámica $T() = O(pq^2)$

4 GitHub

https://github.com/mateonoel2/ProyectoADA