

Speech Compression

Métodos Numéricos Avanzados

Juan Pablo Orsay - 49373
Horacio Miguel Gomez - 50825
Sebastián Andrés Maio - 50386
Federico Bond - 52247
Braulio Sespede - 51074

Noviembre 2016

En éste trabajo se busca analizar la compresión de una imagen en escala de grises de 8-bits mediante el uso de la transformada discreta del coseno en dos dimensiones y codificación mediante el algoritmo de Huffman. A su vez, exploramos distintas formas de aumentar la compresión con pérdida de calidad mediante el uso de filtros luego de haber aplicado DCT.

Índice

1	Procedimiento	3
2	Transformada discreta de coseno	3
3	Compresión	3
3.1	Sin filtro	3
3.2	Filtro moderado	5
3.3	Filtro pesado	6
4	Prueba completa	8
5	Observaciones	10
5.1	Limitaciones	10
5.2	Lena	10
5.3	Mejoras	10
5.3.1	Fast 2D-DCT	10
5.3.2	Algoritmo de Arai-Angui-Nakajima	10
6	Conclusión	10
7	Anexo	11
7.1	Como usar el codigo	11
7.1.1	Instalar	11
7.1.2	Ejecutar	11
7.2	Codigo Fuente	11

1. Procedimiento

Para realizar la compresión seguimos los siguientes pasos:

1. Aplicarle DCT a cada bloque de 8x8 pixeles de la imagen.
2. Codificar los pixeles resultantes usando Huffman.
3. Luego deberíamos guardar la codificación junto con metadata para poder reconstruir la imagen. Ésta etapa se realizará usando el algoritmo especificado en Listing 4.

Para recuperar la imagen, hace falta decodificar el archivo generado previamente y aplicar DCT inversa.

2. Transformada discreta de coseno

Se utilizó la versión sin optimizar tanto de la transformada como de su inversa, y se usaron bloques de 8x8 pixeles, por lo que $N = 8$.

Fórmula de la transformada discreta de coseno[1][4]:

$$F_{u,v} = C_u C_v \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} p_{x,y} \cos \left[\frac{\pi (2x+1) u}{2N} \right] \cos \left[\frac{\pi (2y+1) v}{2N} \right] \quad (1)$$

Fórmula de la transformada discreta inversa de coseno[1]:

$$P_{x,y} = \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} C_u C_v f_{u,v} \cos \left[\frac{\pi (2x+1) u}{2N} \right] \cos \left[\frac{\pi (2y+1) v}{2N} \right] \quad (2)$$

Ambas implementaciones se encuentran en el anexo de éste trabajo bajo las figuras Listing 3 y Listing 6.

3. Compresión

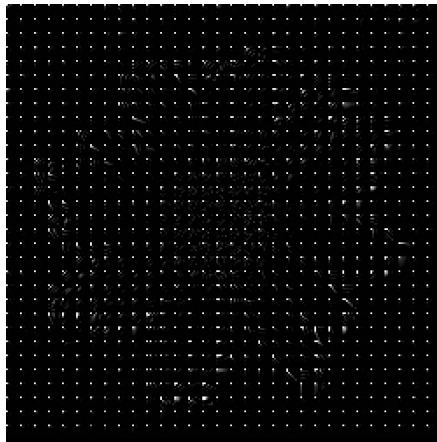
3.1. Sin filtro

Al aplicar la DCT a una imagen prueba y luego pasarla por el algoritmo estimador de compresión, descubrimos que en la siguiente imagen se logra un 21 % de compresión.

```
make flower
env/bin/python src/runner.py source_images/04_flower.png
Running image: source_images/04_flower.png
Compression gain: 20.98 %
```



(a) original



(b) frecuencias



(c) recuperada

Figura 1: Imágenes en el transcurso del algoritmo

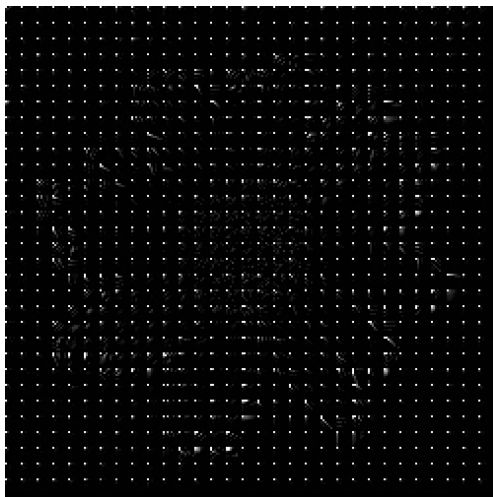
3.2. Filtro moderado

Usando un filtro que retiene únicamente un sector de las frecuencias logramos una mejora en la compresión, pero al reconstruir la imagen utilizando la transformada inversa del coseno podemos notar un deterioro grande de la imagen.

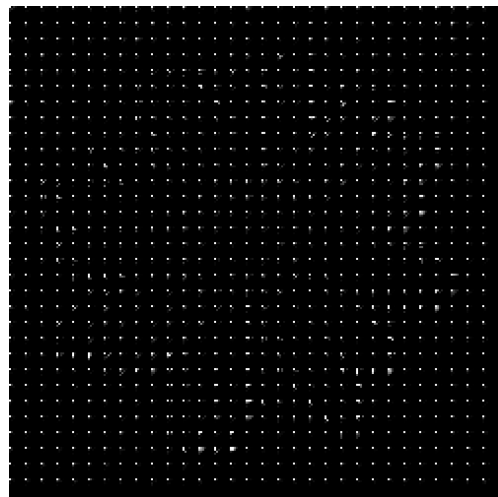
```
1 def filter_moderate(block):
2     result = [
3         [1, 1, 1, 1, 0, 0, 0, 0],
4         [1, 1, 1, 0, 0, 0, 0, 0],
5         [1, 1, 0, 0, 0, 0, 0, 0],
6         [1, 0, 0, 0, 0, 0, 0, 0],
7         [0, 0, 0, 0, 0, 0, 0, 0],
8         [0, 0, 0, 0, 0, 0, 0, 0],
9         [0, 0, 0, 0, 0, 0, 0, 0],
10        [0, 0, 0, 0, 0, 0, 0, 0]
11    ]
12    for x in range(0, BLOCK_SIZE):
13        for y in range(0, BLOCK_SIZE):
14            result[x][y] *= block[x][y]
15    return result
```

Listing 1: Matriz de filtro moderado

```
make flower
env/bin/python src/runner.py source_images/04_flower.png
Running image: source_images/04_flower.png
Compression gain: 27.49 %
```



(a) no filtrada



(b) moderadamente filtrada

Figura 2: Comparación de imágenes de frecuencias



(a) no filtrada



(b) moderadamente filtrada

Figura 3: Comparación de imágenes

3.3. Filtro pesado

Al aplicar un filtro más agresivo (descartar todos salvo un elemento de las frecuencias generadas en cada bloque por el algoritmo de la transformada discreta del coseno) notamos una compresión enorme (92.29 %). Sin embargo, la imagen se encuentra totalmente deteriorada dado que en cada sector de 8x8 hay un solo color.

```

1 def filter_heavy(block):
2     result = [
3         [1, 0, 0, 0, 0, 0, 0, 0],
4         [0, 0, 0, 0, 0, 0, 0, 0],
5         [0, 0, 0, 0, 0, 0, 0, 0],
6         [0, 0, 0, 0, 0, 0, 0, 0],
7         [0, 0, 0, 0, 0, 0, 0, 0],
8         [0, 0, 0, 0, 0, 0, 0, 0],
9         [0, 0, 0, 0, 0, 0, 0, 0],
10        [0, 0, 0, 0, 0, 0, 0, 0]
11    ]
12    for x in range(0, BLOCK_SIZE):
13        for y in range(0, BLOCK_SIZE):
14            result[x][y] *= block[x][y]
15    return result

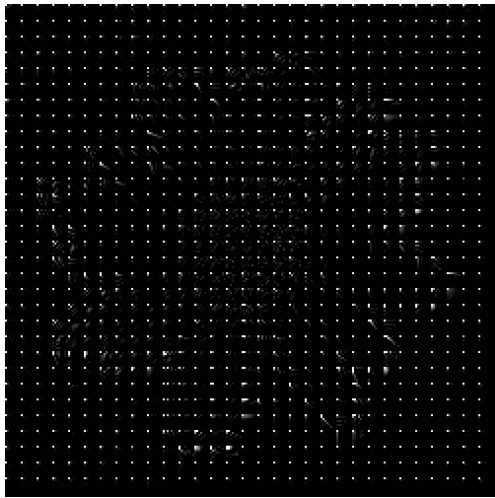
```

Listing 2: Matriz de filtro pesado

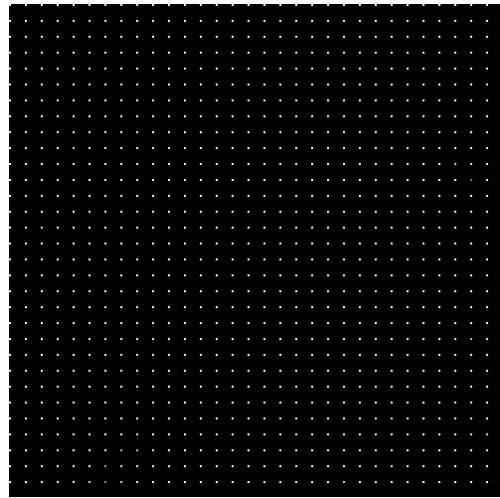
```

make flower
env/bin/python src/runner.py source_images/04_flower.png
Running image: source_images/04_flower.png
Compression gain: 92.29 %

```



(a) no filtrada

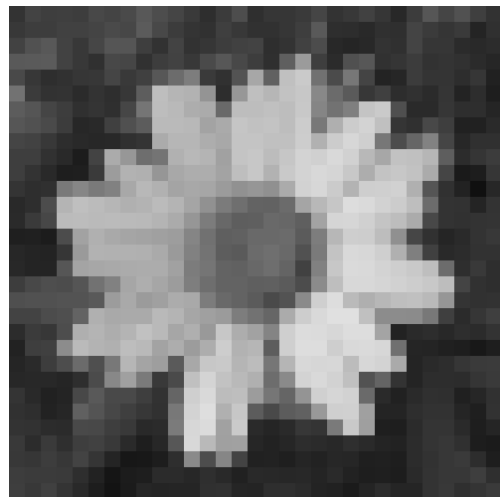


(b) muy filtrada

Figura 4: Comparacion de imagenes de frecuencias



(a) no filtrada



(b) muy filtrada

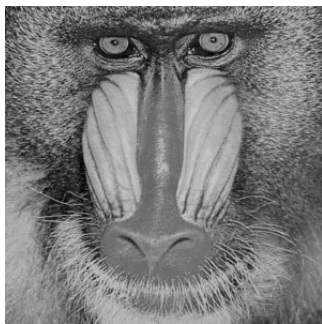
Figura 5: Comparación de imágenes

4. Prueba completa

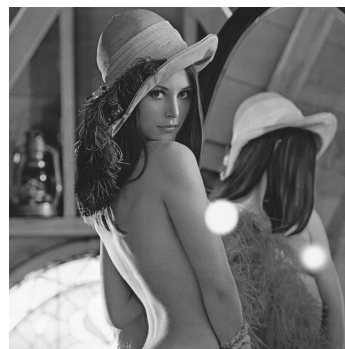
A continuación mostramos el resultado del algoritmo de compresión al ser ejecutado sobre un set de 10 imágenes.

El promedio de compresión fue de aproximadamente 13% y hemos tenido algunos casos en los cuales la imagen comprimida resultó ser de mayor tamaño que la original (aquellos en los que la compresión es negativa).

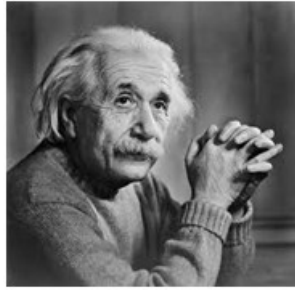
```
make all
env/bin/python src/runner.py all
Running image: source_images/01_mandrill.png
Compression gain: 20.34 %
Running image: source_images/02_lena.png
Compression gain: -4.34 %
Running image: source_images/03_einstein.png
Compression gain: 17.31 %
Running image: source_images/04_flower.png
Compression gain: 20.99 %
Running image: source_images/05_loro.png
Compression gain: 14.45 %
Running image: source_images/06_lena_face.png
Compression gain: -11.93 %
Running image: source_images/07_puppy.png
Compression gain: 34.93 %
Running image: source_images/08_random.png
Compression gain: -0.47 %
Running image: source_images/09_picaflor.png
Compression gain: 20.61 %
Running image: source_images/10_bebe.png
Compression gain: 1.17 %
```



(a) 01 mandril



(b) 02 lena



(a) 03 einstein



(b) 04 flower



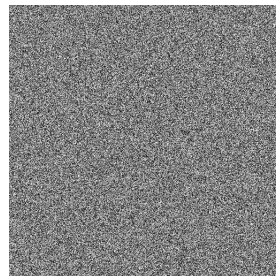
(c) 05 loro



(d) 06 lena face



(e) 07 puppy



(f) 08 random



(g) 09 picaflor



(h) 10 bebe

5. Observaciones

5.1. Limitaciones

Descubrimos que debido a un problema con el algoritmo implementado, aquellas imágenes cuyas dimensiones no sean múltiplos de 8 contienen pérdida de información en la región izquierda e inferior.

A su vez, a la hora de querer recuperar una imagen codificada, descubrimos que nuestra implementación de la transformada inversa discreta del coseno es bastante lenta, tardando cerca de 20 segundos para recuperar la información de una imagen de aproximadamente 512x512 píxeles.

5.2. Lena

Lena es una imagen que se utiliza recurrentemente en el análisis de algoritmos de procesamiento de imágenes[5] debido a su detalle, sombreado, regiones de color planas y texturas.

Hemos descubierto que en ciertos casos (principalmente en fotos de Lena) el método de compresión genera imágenes de mayor peso que las originales, haciendo la compresión inefectiva. En la sección anterior podemos ver éste fenómeno.

5.3. Mejoras

Si bien codificamos la implementación original del algoritmo DCT, encontramos que existen variaciones sobre el algoritmo de la transformada discreta del coseno:

5.3.1. Fast 2D-DCT

Se puede implementar DCT en 2 dimensiones aplicando el algoritmo DCT a todas las filas de cada bloque y luego aplicar DCT a cada columna del bloque resultante. La complejidad de dicho algoritmo disminuye de $O(n^4)$ a $O(n^3)$ [2].

5.3.2. Algoritmo de Arai-Angui-Nakajima

La idea es la misma que la del algoritmo anterior pero se reemplaza la implementación de la transformada discreta del coseno en 1 dimensión por una implementación similar a la transformada rápida de Fourier resultando en un algoritmo de complejidad $O(n \log n)$. En nuestro trabajo, usamos ésta implementación para generar el reporte completo de compresiones ya que usando el original nos tomaba mucho tiempo. Para generar el reporte individual, usamos la implementación original[3][2].

6. Conclusión

Mediante el uso de la transformada de coseno y el algoritmo de codificación de Huffman hemos podido comprobar, dentro del conjunto de imágenes que conseguimos, que se puede

comprimir (sin perdida de calidad) utilizando transformadas. La principal ventaja de utilizar DCT es que se "juntan" los datos de imagen de un bloque en un número casi óptimo de coeficientes descorrelacionados, dando lugar a una compresión significativa en algoritmos de codificación aritmética como Huffman.

La variación que propusimos es aplicar un filtro, que si bien se pierde calidad en imagen final, luego de aplicar la antitransformada para conseguir la imagen original nuevamente, se gana una compresión que es inversamente proporcional al número de frecuencias filtradas.

7. Anexo

7.1. Como usar el código

7.1.1. Instalar

```
git clone git@bitbucket.org:iLNaNo/mna-fft.git
cd mna-fft
make install
```

7.1.2. Ejecutar

```
make all
make flower
make ...
```

7.2. Código Fuente

```
1 def dct_2d(pixels, offset_x, offset_y):
2     out = array2d(BLOCK_SIZE, BLOCK_SIZE)
3     for u in range(0, BLOCK_SIZE):
4         for v in range(0, BLOCK_SIZE):
5             z = 0.0
6             cu, cv = coeffs(u, v, BLOCK_SIZE)
7             for x in range(0, BLOCK_SIZE):
8                 for y in range(0, BLOCK_SIZE):
9                     pixel = pixels[x + offset_x, y + offset_y]
10                    val = pixel * \
11                        cos(pi * (2 * x + 1) * u / (2.0 * BLOCK_SIZE)) *
12                        \
13                        cos(pi * (2 * y + 1) * v / (2.0 * BLOCK_SIZE))
14                    z = z + val
15                z = cu * cv * z
16                out[u][v] = z
17     return quantize(out)
```

Listing 3: DCT

```

1 def nextpow2(i):
2     n = 1
3     while n < i:
4         n *= 2
5     return n
6
7
8 def complen(image_data):
9     symb2count = defaultdict(int)
10    symb2freq = defaultdict(float)
11    for pixel in image_data:
12        symb2count[pixel] += 1
13    total_size = len(image_data)
14    for pixel in image_data:
15        symb2freq[pixel] = symb2count[pixel] / total_size
16    huff = huffman(symb2count)
17
18    lencomp = 0
19    symbols = list(symb2count.keys())
20
21    # adding dictionary
22    lencomp += nextpow2(max(symbols)) * len(symbols)
23    for p in huff:
24        lencomp += len(p[1])
25
26    # adding stored width and height
27    lencomp += 2 * 32
28
29    # adding compressed image
30    for p in huff:
31        lencomp += symb2freq[p[0]] * len(p[1])
32
33    return lencomp

```

Listing 4: Función complen

```

1 def huffman(symb2freq):
2     heap = [[wt, [sym, ""]] for sym, wt in symb2freq.items()]
3     heapify(heap)
4     while len(heap) > 1:
5         lo = heappop(heap)
6         hi = heappop(heap)
7         for pair in lo[1:]:
8             pair[1] = '0' + pair[1]
9         for pair in hi[1:]:
10            pair[1] = '1' + pair[1]
11        heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])
12    return sorted(heappop(heap)[1:], key=lambda p: (len(p[-1]), p))

```

Listing 5: Huffman

```

1  def idct_2d(dct_data, offset_x, offset_y):
2      out = array2d(BLOCK_SIZE, BLOCK_SIZE)
3      for x in range(0, BLOCK_SIZE):
4          for y in range(0, BLOCK_SIZE):
5              z = 0.0
6              for u in range(0, BLOCK_SIZE):
7                  for v in range(0, BLOCK_SIZE):
8                      cu, cv = coeffs(u, v, BLOCK_SIZE)
9                      pixel = dct_data[u][v]
10                     val = cu * cv * pixel * \
11                         cos((2.0 * x + 1) * u * pi / (2 * BLOCK_SIZE)) *
12                         \
13                         cos((2.0 * y + 1) * v * pi / (2 * BLOCK_SIZE))
14                     z = z + val
15             out[x][y] = z
16     return quantize(out)

```

Listing 6: DCT inversa

```

1  def quantize(data):
2      for y in range(0, BLOCK_SIZE):
3          for x in range(0, BLOCK_SIZE):
4              data[x][y] = int(max(min(round(data[x][y]), 255.0), 0.0))
5      return data

```

Listing 7: (I)DCT redondeado

Referencias

- [1] *Discrete Consine Transform*. URL: <http://www.mathworks.com/help/images/discrete-cosine-transform.html> (visitado 10-11-2015).
- [2] *Discrete Cosine Transform*. URL: <https://unix4lyfe.org/dct/> (visitado 10-11-2015).
- [3] *Fast Discrete Consine Transform. AAN*. URL: <https://unix4lyfe.org/dct/listing3.c> (visitado 10-11-2015).
- [4] *Wikipedia - Discrete Cosine Transform*. URL: https://en.wikipedia.org/wiki/Discrete_cosine_transform (visitado 10-11-2015).
- [5] *Wikipedia - Lenna*. URL: <https://en.wikipedia.org/wiki/Lenna> (visitado 11-11-2015).