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 este trabajo implementaremos un simple compresor del habla. Nos basaremos en *2nd International Conference on Computer and Communication Technology*, donde se propone un esquema muy simple de compresión basado en tres pasos: transformación, cuantificación y codificación Huffman.

Índice

1. Procedimiento

Para realizar la compresión seguimos los siguientes pasos:

1. Obtener la FFT de la muestra de audio.
2. De los coeficientes obtenidos se guarda la mitad +1.
3. Los coeficientes cuyo valor absoluto es menor que *Epsilon* se los hace 0.
4. Cuantificamos la parte real y la imaginaria de los coeficientes con L bits.
5. Codificamos con el algoritmo de Huffman los coeficientes.

Para recuperar la grabación, hace falta de-codificar la información generada previamente y aplicar IFFT inversa.

2. Ejemplo compresión y descompresión

Para realizar la compresión y descompresión de un archivo de sonido, abrir octave dentro del directorio *src* del proyecto y ejecutar lo siguiente:

```
wav_name = "01"
epsilon = 0.1
L = 16
[compressed, scale] = compress(wav_name, epsilon, L)
```

Siendo:

1. wav_name el nombre del archivo dentro de resources/wav a comprimir
2. *epsilon* el valor a utilizar para eliminar ruido
3. L el número de bits a utilizar en la quantización

En *compressed* obtenemos el wav comprimido, cuantizado y truncado que luego puede ser descomprimido mediante el siguiente código:

```
wav_name = "01" %01_recompressed.wav
uncompressed(compressed, "01")
```

Tras

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

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

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

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 %
```

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.

```
def filter_moderate(block):
    result = [
        [1, 1, 1, 1, 0, 0, 0, 0],
        [1, 1, 1, 0, 0, 0, 0, 0],
        [1, 1, 0, 0, 0, 0, 0, 0],
        [1, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0]
    ]
    for x in range(0, BLOCK_SIZE):
        for y in range(0, BLOCK_SIZE):
            result[x][y] *= block[x][y]
    return result
```

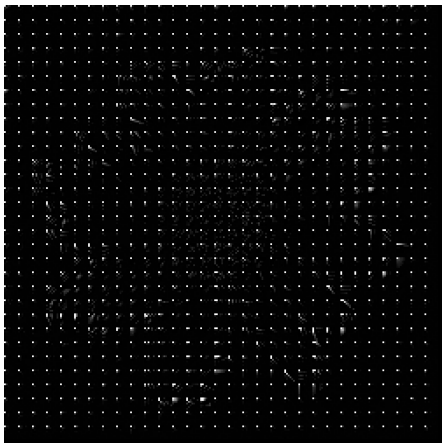
Listing 1: Matriz de filtro moderado

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

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.

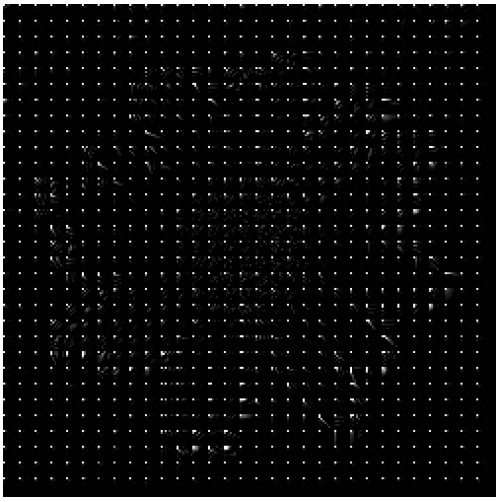
(a) original



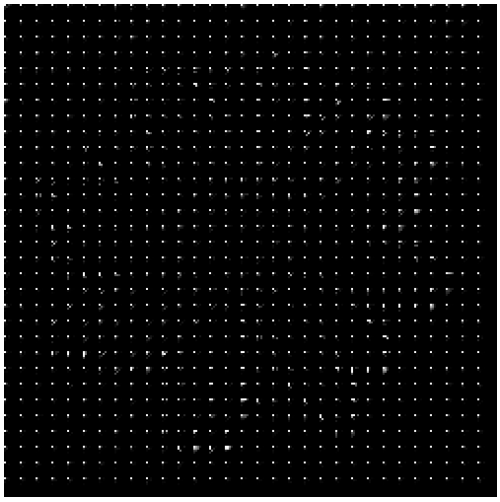
(b) frecuencias

(c) recuperada

Figura 1: Imágenes en el transcurso del algoritmo



(a) no filtrada



(b) moderadamente filtrada

Figura 2: Comparación de imágenes de frecuencias

(a) no filtrada

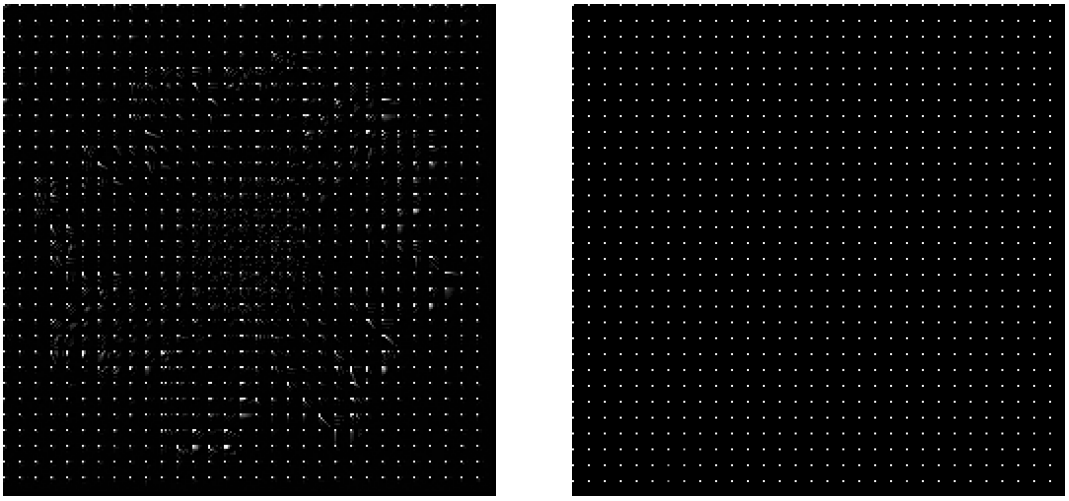
(b) moderadamente filtrada

Figura 3: Comparación de imágenes

```
def filter_heavy(block):
    result = [
        [1, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0]
    ]
    for x in range(0, BLOCK_SIZE):
        for y in range(0, BLOCK_SIZE):
            result[x][y] *= block[x][y]
    return result
```

Listing 2: Matriz de filtro pesado

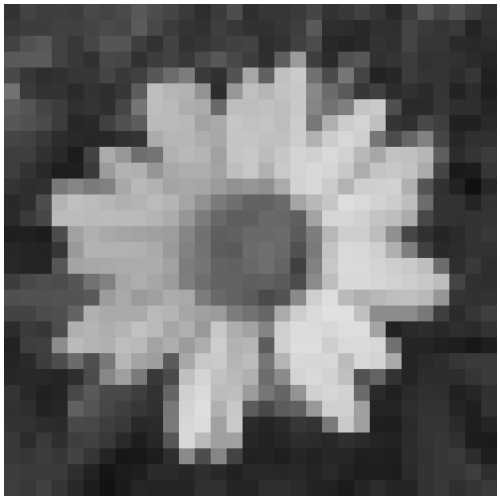
```
make flower
env/bin/python src/runner.py source_images/04_flower.png
Running image:  source_images/04_flower.png
Compression gain: 92.29 %
```



(a) no filtrada (b) muy filtrada

Figura 4: Comparacion de imagenes de frecuencias

(a) no filtrada



(b) muy filtrada

Figura 5: Comparación de imágenes

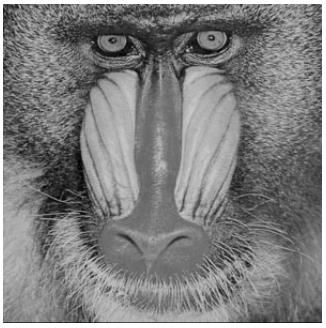
4. Prueba completa

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).

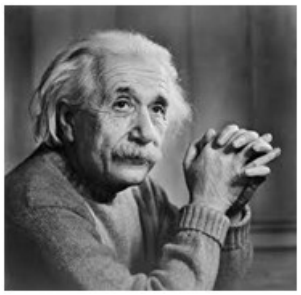
```
make all
env/bin/python src/runner.py all
Running image:  source_images/01_mandril.png
Compression gain: 20.34 %
Running image:  source_images/02_lena.png
Compression gain: -4.34 %
Running image:  source_images/03_einstein.png
Compression gain: 17.31 %
Running image:  source_images/04_flower.png
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



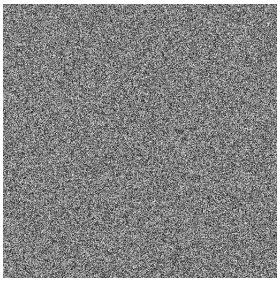
(a) 03 einstein

(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 pixeles.

5.2. Lena

Lena es una imagen que se utiliza recurrentemente en el análisis de algoritmos de procesamiento de imágenes[4] 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 codificacion aritmetica como Huffman.

La variacion 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 compresion que es inversamente proporcional al numero de frecuencias filtradas.

7. Anexo

7.1. Como usar el codigo

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. Codigo Fuente

```
def dct_2d(pixels, offset_x, offset_y):
    out = array2d(BLOCK_SIZE, BLOCK_SIZE)
    for u in range(0, BLOCK_SIZE):
        for v in range(0, BLOCK_SIZE):
            z = 0.0
            cu, cv = coeffs(u, v, BLOCK_SIZE)
            for x in range(0, BLOCK_SIZE):
                for y in range(0, BLOCK_SIZE):
                    pixel = pixels[x + offset_x, y + offset_y]
                    val = pixel * \
                        cos(pi * (2 * x + 1) * u / (2.0 * BLOCK_SIZE)) *
                            \
                        cos(pi * (2 * y + 1) * v / (2.0 * BLOCK_SIZE))
                    z = z + val
            z = cu * cv * z
            out[u][v] = z
    return quantize(out)
```

Listing 3: DCT

```
def nextpow2(i):
    n = 1
    while n < i:
        n *= 2
    return n


def complen(image_data):
    symb2count = defaultdict(int)
    symb2freq = defaultdict(float)
    for pixel in image_data:
        symb2count[pixel] += 1
    total_size = len(image_data)
    for pixel in image_data:
        symb2freq[pixel] = symb2count[pixel] / total_size
    huff = huffman(symb2count)

    lencomp = 0
    symbols = list(symb2count.keys())

    # adding dictionary
    lencomp += nextpow2(max(symbols)) * len(symbols)
    for p in huff:
        lencomp += len(p[1])

    # adding stored width and height
    lencomp += 2 * 32

    # adding compressed image
    for p in huff:
        lencomp += symb2freq[p[0]] * len(p[1])

    return lencomp
```

Listing 4: Función complen

```
def huffman(symb2freq):
    heap = [[wt, [sym, ""]] for sym, wt in symb2freq.items()]
    heapify(heap)
    while len(heap) > 1:
        lo = heappop(heap)
        hi = heappop(heap)
        for pair in lo[1:]:
            pair[1] = '0' + pair[1]
        for pair in hi[1:]:
            pair[1] = '1' + pair[1]
        heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])
    return sorted(heappop(heap)[1:], key=lambda p: (len(p[-1]), p))
```

Listing 5: Huffman

```
def idct_2d(dct_data, offset_x, offset_y):
    out = array2d(BLOCK_SIZE, BLOCK_SIZE)
    for x in range(0, BLOCK_SIZE):
        for y in range(0, BLOCK_SIZE):
            z = 0.0
            for u in range(0, BLOCK_SIZE):
                for v in range(0, BLOCK_SIZE):
                    cu, cv = coeffs(u, v, BLOCK_SIZE)
                    pixel = dct_data[u][v]
                    val = cu * cv * pixel * \
                        cos((2.0 * x + 1) * u * pi / (2 * BLOCK_SIZE)) * \
                        cos((2.0 * y + 1) * v * pi / (2 * BLOCK_SIZE))
                    z = z + val
            out[x][y] = z
    return quantize(out)
```

Listing 6: DCT inversa

```
def quantize(data):
    for y in range(0, BLOCK_SIZE):
        for x in range(0, BLOCK_SIZE):
            data[x][y] = int(max(min(round(data[x][y]), 255.0), 0.0))
    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 - Lenna.* URL: https://en.wikipedia.org/wiki/Lenna (visitado 11-11-2015).