INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

ÁREA DEPARTAMENTAMENTAL DE ENGENHARIA DE ELECTRÓNICA E TELECOMUNICAÇÕES E DE COMPUTADORES

Licenciatura em Engenharia Informática e Multimédia

Codificação de Sinais Multimédia

RELATÓRIO TP4

<2017 - 2018>

ALUNOS:

39758 André Fonseca

33104, Tiago Oliveira

1 Introdução

Neste último trabalho prático, serão implementados três codificadores de vídeo com base nos codificadores estudados nas aulas.

A ideia fundamental da codificação de vídeo nasce da necessidade de comprimir conteúdos que possuem uma grande quantidade de informação, para que possam ser mais rapidamente processados, armazenados ou até mesmo partilhados.

Os algoritmos de compressão de vídeo exploram principalmente, o facto de que na maior parte dos casos existir pouco movimento nos objetos que compõe a cena. Se considerarmos que um vídeo num formato standard é reproduzido entre 24 a 30 frames por segundo, e analisarmos algumas amostras de frames, depressa se percebe que na maior parte do tempo a cena permanece a mesma e apenas alguns objetos se movimentam. Assim ao invés de enviar ou processar todas as frames de igual forma, podemos criar algoritmos que processam apenas a diferença entre cada frame enviando apenas os objetos que se movem.

Será também possível haver predição do movimento dos objetos contidos na cena e por consequência comprimir ainda mais a informação processada.

2 Desenvolvimento

Dado que um vídeo é um conjunto de imagens, é possível numa primeira fase aplicar os conceitos aprendidos no trabalho pratico 3, e comprimir cada frame usando uma codificação JPEG.

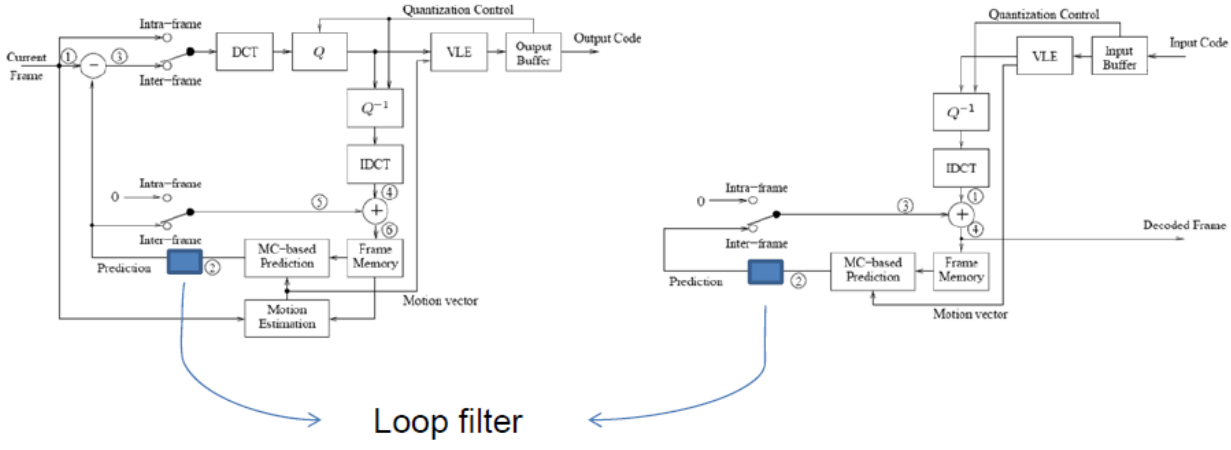
Posteriormente será feita uma conversão do espaço de cores de RGB para YCbCr de maneira a que se consiga extrair a informação da luminosidade para um componente separado dos componentes de cor, usando assim as limitações do cérbero a nosso favor.

Será feito um chroma-subsampling para reduzir ainda mais a entropia das frames.

A codificação das frames que constituem o vídeo será feita em termos espaciais, usando os processos descritos em cima, e em termos temporais, isto é, irá ser feita a diferença entre frames, removendo assim a redundância temporal. Isto acontece porque ao analisar a maior parte dos vídeos percebe-se que na maior parte do tempo existe uma grande quantidade de pixéis que se mantém inalterados ao longo da sequência de frames. Um exemplo clássico deste acontecimento, é quando é feita uma comunicação via webcam, onde a imagem permanece inalterada ao longo do tempo com exceção do movimento da pessoa que está a ser gravada.

Isto deixa de ser verdade quando a camara se move ou a quantidade de movimento é tão alto que deixa de ser viável processar as frames de um ponto de vista de compressão, e é preferível enviar ou processar a frame completa. Neste caso, o processo de compressão é reiniciado, e a Intraframe será atualizada.

Este processo de seleção de Intraframes é facilmente explicado observando o esquema algorítmico do codificador/descodificador:



É evidente no esquema mostrado, como o algoritmo irá gerar as Interframes, isto é, as frames a serem enviadas para o output, à custa da Interframe anterior e da Intraframe corrente.

Ao processar a Interframe usando a transformada DCT e quantizando o resultado para diminuir a gama de variação do valor dos pixeis, é realizado o calculo do movimento predito para a proxima frame, resultando num vector que irá conter a informação da posição original do objecto na imagem, bem como a localização futura deste. Este passo é fundamental para garantir uma compressão que não só elimina informação reduntante da imagem, mas que também consegue estimar/compensar o movimento relaizado por um determinado objecto na cena.

Esta codificação é feita, à semelhança do que acontece com algoritmo JPEG, por blocos de 16x16 para o caso da frame que contem a informação da luminancia e blocos de 8x8 para as frames que contém a crominancia.

Esta é um passo fundamental pois não seria possivel fazer a deteção de movimento caso a imagem não fosse analisada em pequenos blocos pois a deteção de movimento é relaizada usando uma procura exaustiva em cada bloco, que irá usar o erro quadratico médio numa serie de calculos utilizando o bloco atual a ser analisado, e uma janela de pesquisa (-15 a 15 pixeis) e onde será calculado o vector de movimento ótimo.

2.1 Implementação

Podemos observar a implementação da pesquisa exaustiva que devolve o bloco que produz o menor erro quadratico possivel:

```
def find_similar(self, block: Block, layer: Layer) -> (Block, float):
    # Original block coordinates
    r, c = block.origin[0]

    pixels: ndarray = layer.pixels
    block_size = layer.blocks[0, 0].size

    total_errors = power(2 * Exhaustive.SEARCH_WINDOW + 2, 2)
    errors = array([inf] * total_errors).reshape((int(sqrt(total_errors)), int(sqrt(total_errors))))

    blocks = array([Block] * total_errors).reshape((int(sqrt(total_errors)), int(sqrt(total_errors))))

    # Indexes for errors and blocks
    r2 = -1
    c2 = -1

    for r1 in range(r + -1 * Exhaustive.SEARCH_WINDOW, r + Exhaustive.SEARCH_WINDOW):
        r2 += 1
        if r1 < 0:
            continue
        r3 = r1 + block_size
        if r3 > pixels.shape[0]:
            continue
        for c1 in range(c + -1 * Exhaustive.SEARCH_WINDOW, c + Exhaustive.SEARCH_WINDOW):
            c2 += 1
            if c1 < 0:
                continue
            c3 = c1 + block_size
            if c3 > pixels.shape[1]:
                continue

            blocks[r2, c2] = Block(pixels[r1: r3, c1: c3], (r1, c1), (r2, c2), block_size)
            errors[r2, c2] = self.calculate_error(blocks[r2, c2], block)

        c2 = -1

    r, c, min_error = self._find_smallest_error(errors)
    return blocks[r, c], min_error
```

Este método será chamado pelo codificador para calcular o vector de movimento bem como a frame predita a ser usada como proxima Interframe, e a diferença entre frames (erro) que será codificadoe processado para ser mais tarde descodificado. Podemos observar a implementação do algoritmo de codificação:

```
def encode(self, previous_frame: YCbCrFrame, frame: YCbCrFrame):
    layers = frame.layers
    previous_layers = previous_frame.layers
    predicted_image = zeros(frame.size, dtype=int).reshape(frame.shape)

    # List of origin and target vectors
    vectors = array([Vectors] * frame.layers.size)

    for l in range(previous_layers.shape[0]):
        vectors[l] = Vectors((
            zeros(2 * 2 * previous_layers[l].blocks.size, dtype=int)
        ).reshape(
            (previous_layers[l].blocks.size, 2 * 2)
        ))

        previous_blocks = previous_layers[l].blocks

        for r in range(previous_blocks.shape[0]):
            for c in range(previous_blocks.shape[1]):
                similar_block: Block = self.__search_strategy.find_similar(previous_blocks[r, c], layers[l])[0]
                r2, c2 = similar_block.origin[0]

                # Predict the block
                r3 = r2 + similar_block.size
                c3 = c2 + similar_block.size

                previous_blocks[r, c].target = r2, c2
                predicted_image[r2: r3, c2: c3, l] = previous_blocks[r, c].pixels

                # Collect origin and target vector
                r4 = r * previous_blocks.shape[1] + c
                vectors[l][r4, 0: 2] = previous_blocks[r, c].origin[0]
                vectors[l][r4, 2: 4] = previous_blocks[r, c].target

    error_frame = YCbCrFrame(frame.pixels - predicted_image, -1 * frame.index)
    predicted_frame = YCbCrFrame(predicted_image, frame.index)

    return predicted_frame, vectors, error_frame
```