

Relatório

Introdução

Neste Relatório são abordados todos os exercícios realizados do Projeto #02.

São utilizadas bibliotecas como AudioFile e OpenCV. Os exercícios da parte C que utilizam a biblioteca OpenCV necessitam que esta biblioteca esteja instalada bem como o CMake.

Código do Projeto: https://github.com/miguelf18/IC_Proj2.

Para testar os programas das partes A e B deve-se executar o comando **make** no diretório do exercício e de seguida correr o ficheiro executável **.exe**.

Para testar os programas da parte C, deve-se ir para o diretório do exercício e fazer **Build**, após a operação estar concluída, deve-se executar o comando **cd build\Debug** para de seguida correr o ficheiro executável **.exe**.

Parte A

Exercícios 1 e 2 (BitStream) - Antes de se construir os codecs pedidos, foram criadas classes de suporte para facilitar a codificação/descodificação. Uma dessas classes é a Bitstream que está encarregue de escrever/ler informação bit a bit para/de um ficheiro. No nosso caso, a bitstream pode ler/escrever 1 bit de cada vez ou vários ao mesmo tempo e também trata da parte de mais baixo nível referente à abertura ou fecho do ficheiro de dados. No entanto, como em c++ só é possível escrever/ler 1 byte de dados de cada vez nos ficheiros, a bitstream utiliza um buffer interno para armazenar o byte e portanto, sempre que o buffer estiver vazio ou cheio, é efetuada a operação necessária para escrever o conteúdo do buffer para o ficheiro, ou ler o próximo byte de dados para o mesmo buffer. Note-se que no caso da escrita, se os últimos bits não formarem um byte completo então tem que existir um padding para encher o resto do buffer da bitstream e, posteriormente, escrever o byte de dados no ficheiro, sendo que o programa que utilizar a bitstream é que está encarregue de realizar este padding. Já no caso da leitura, a bitstream devolve uma flag que indica se ocorreu algum erro no ficheiro ou se o end of file foi alcançado.

Finalmente, juntamente com a classe criada também se incluiu um pequeno programa para testar as diversas operações onde os resultados pretendidos são observados.

Exercícios 3 e 4 (Golomb) - Antes de se construir os codecs pedidos, foram criadas classes de suporte para facilitar a codificação/descodificação. Além da classe bitstream referida anteriormente nos exercícios 1 e 2, foi também criada a classe Golomb com o objetivo de implementar um método para codificar números e outro para decodificá-los, sendo possível especificar o valor do parâmetro m .

Os códigos de Golomb baseiam-se na separação de um inteiro em 2 partes, em que uma dessas partes é representada através de código unário e outra através de código binário.

Os códigos de Golomb dependem de um parâmetro m , sendo m um valor inteiro maior que 0. Tendo um inteiro $i \geq 0$:

$$q = \lfloor \frac{i}{m} \rfloor, \quad \text{em que } q \text{ é representado em código unário.}$$

De seguida, caso o valor de m especificado seja uma potência de 2, um inteiro $m \geq 0$ é representado por 2 números q e r , em que:

$$r = i - qm, \quad \text{sendo } r \text{ representado em código binário.}$$

Caso o valor de m especificado não seja uma potência de 2, então é usada a representação em código binário truncado. Para tal, a primeira coisa a fazer é calcular b , em que:

$$b = \lceil \log_2 m \rceil$$

Sabendo o valor de b , já é possível fazer a codificação, que é feita da seguinte forma:

- Codificar os primeiros $2^b - m$ valores de r utilizando os primeiros $2^b - m$ códigos binários com $b - 1$ bits.
- Codificar os restantes valores de r codificando o número $r + 2^b - m$ em binário com b bits.

Utilizando estas fórmulas foram desenvolvidos um codificador juntando os códigos unário e binário obtidos e um decodificador.

Finalmente, juntamente com a classe criada, também se incluiu um pequeno programa para testar a classe criada.

Parte B

Exercício 1 - Relativamente à parte B, o objetivo é implementar um lossless codec de áudio baseado em predictive coding seguido de entropy coding através da classe Golomb já construída nos exercícios anteriores. O codec implementado explora não só redundâncias temporais através do predictive coding, mas também redundâncias de canais, codificando apenas um só canal que resulta da média entre os canais da esquerda e direita visto que os valores dos dois canais são semelhantes. Em termos de predictive coding, o codec faz uso dos seguintes preditores polinomiais:

$$\begin{aligned}\gamma_n^{(0)} &= 0 \\ \gamma_n^{(1)} &= x_{n-1} \\ \gamma_n^{(2)} &= 2x_{n-1} - x_{n-2} \\ \gamma_n^{(3)} &= 3x_{n-1} - 3x_{n-2} + x_{n-3}\end{aligned}$$

Sendo que os valores de γ_{nb} correspondem às estimativas dos valores de áudio, e os valores x_n correspondem aos valores reais do ficheiro de áudio. Com base nos valores estimados são então calculados os valores residuais, referentes ao erro entre os valores estimados e os verdadeiros, através da seguinte fórmula:

$$r_n = x_n - \gamma_n$$

Estes valores residuais são então codificados através do Golomb, juntamente com alguma informação do ficheiro de áudio, i.e., número de samples, número de canais, etc..... Para além disto, também é codificado o sinal do valor com 1 bit extra, caso não seja 0, para distinguir os negativos dos positivos.

No entanto, como o Golomb só funciona para valores inteiros, ainda foi necessário converter os valores de áudio, que, com o uso da biblioteca AudioFile, se encontram em formato de float, em inteiros, antes de calcular os valores residuais.

No caso da descodificação, só é necessário realizar o processo inverso à codificação, tendo o cuidado de reverter os valores de áudio de volta para float. Note-se que tanto o codificador como o descodificador necessitam do mesmo parâmetro de Golomb, m , para funcionarem corretamente. No final, observou-se os resultados pretendidos, ou seja, o ficheiro de áudio original foi comprimido para um novo ficheiro de tamanho significativamente reduzido, seguido da sua descodificação onde os valores originais foram reconstruídos. Para verificar a influência do parâmetro de Golomb no tamanho do ficheiro comprimido e tempo de execução, utilizou-se diferentes valores, juntamente com os ficheiros wav fornecidos, e os resultados obtidos podem ser vistos na **Tabela 1 (tamanho do ficheiro comprimido / tempo de execução)** para a compressão e na **Tabela 2 (tempo de execução)** para a descompressão. Conclui-se que o valor ótimo para o m estará dependente dos valores a serem codificados e que, neste caso, na maior parte dos ficheiros, um m pequeno produz os melhores resultados.

	sample01	sample02	sample03	sample04	sample05	sample06	sample07
m=2	832,7 Kb / 0,43 sec	473,9 Kb / 0,22 sec	472,1 Kb / 0,28 sec	283,3 Kb / 0,18 sec	408,1 Kb / 0,27 sec	454,9 Kb / 0,31 sec	1,4 Mb / 0,40 sec
m=5	786,1 Kb / 0,43 sec	411,0 Kb / 0,21 sec	484,8 Kb / 0,28 sec	309,0 Kb / 0,18 sec	456,0 Kb / 0,27 sec	513,1 Kb / 0,32 sec	888,7 Kb / 0,35 sec
m=8	836,7 Kb / 0,43 sec	434,6 Kb / 0,21 sec	550,5 Kb / 0,28 sec	360,5 Kb / 0,19 sec	546,8 Kb / 0,29 sec	629,8 Kb / 0,33 sec	803,1 Kb / 0,34 sec
m=11	863,5 Kb / 0,44 sec	436,5 Kb / 0,22 sec	555,2 Kb / 0,29 sec	360,2 Kb / 0,19 sec	544,6 Kb / 0,29 sec	625,1 Kb / 0,33 sec	781,2 Kb / 0,35 sec
m=14	924,9 Kb / 0,44 sec	460,2 Kb / 0,22 sec	593,7 Kb / 0,30 sec	388,5 Kb / 0,19 sec	578,7 Kb / 0,30 sec	659,6 Kb / 0,34 sec	784,9 Kb / 0,35 sec
m=17	968,7 Kb / 0,45 sec	489,0 Kb / 0,22 sec	646,8 Kb / 0,30 sec	428,7 Kb / 0,20 sec	652,7 Kb / 0,30 sec	753,0 Kb / 0,35 sec	791,1 Kb / 0,34 sec
m=20	971,1 Kb / 0,45 sec	489,9 Kb / 0,22 sec	647,0 Kb / 0,30 sec	428,0 Kb / 0,20 sec	651,7 Kb / 0,30 sec	751,8 Kb / 0,35 sec	792,1 Kb / 0,34 sec

Tabela 1. Influência do parâmetro de Golomb na compressão

	sample01	sample02	sample03	sample04	sample05	sample06	sample07
m=2	0,56 sec	0,29 sec	0,37 sec	0,24 sec	0,37 sec	0,43 sec	0,50 sec
m=5	0,56 sec	0,28 sec	0,37 sec	0,24 sec	0,37 sec	0,43 sec	0,45 sec
m=8	0,57 sec	0,28 sec	0,38 sec	0,25 sec	0,39 sec	0,45 sec	0,45 sec
m=11	0,57 sec	0,28 sec	0,38 sec	0,25 sec	0,39 sec	0,45 sec	0,45 sec
m=14	0,58 sec	0,29 sec	0,39 sec	0,25 sec	0,39 sec	0,45 sec	0,45 sec
m=17	0,59 sec	0,29 sec	0,40 sec	0,26 sec	0,40 sec	0,47 sec	0,45 sec
m=20	0,60 sec	0,29 sec	0,40 sec	0,26 sec	0,40 sec	0,47 sec	0,45 sec

Tabela 2. Influência do parâmetro de Golomb na descompressão

Exercício 2 - Expandindo no codec de áudio criado, implementou-se uma nova função para calcular os histogramas e respectivas entropias dos valores de áudio originais e dos valores residuais calculados. A comparação das entropias podem ser vistas na **Tabela 3**. Conclui-se, então, que com os métodos usados para a codificação, é possível representar a mesma quantidade de informação utilizando menos bits, resultando num ficheiro de tamanho reduzido com entropia inferior. Relembrando que o cálculo da entropia segue a seguinte fórmula:

$$H(samples) = - \sum_{i=1}^n P(samples(i)) * \log_2(P(samples(i)))$$

	sample01	sample02	sample03	sample04	sample05	sample06	sample07
original	6.90553	5.82002	6.20988	6.93575	5.64616	5.24761	8.10531
residual	4.42043	4.61915	3.69357	3.09668	2.76141	2.39792	6.41462

Tabela 3. Comparação entre entropia original e entropia residual

Exercício 3 - Finalmente, alterou-se o codec de áudio para que ocorresse a codificação de uma forma lossy ao contrário de lossless. Para este efeito, recorreu-se à quantização escalar uniforme dos samples de áudio, em que são descartados bits nos valores de sample originais da seguinte forma:

$$(sample \gg nbits) \ll nbits$$

Onde *nbits* é um parâmetro que pode ser ajustado. Os resultados da entropia dos novos valores residuais quantizados podem ser vistos na **Tabela 4**. Conclui-se que com algum método de lossy coding obtém-se uma entropia mais reduzida comparado com o lossless coding, no entanto, é introduzido ruído no ficheiro de áudio decodificado e, portanto, é necessário encontrar os parâmetros ideais para comprimir um ficheiro de áudio o máximo possível com o menor erro possível introduzido no ficheiro decodificado. Naturalmente, também se terá que ajustar o parâmetro de Golomb que produz os melhores resultados para os novos valores quantizados. No nosso caso, visto que a maior parte dos valores de áudio são pequenos, o *nbits* também deverá ser pequeno para os melhores resultados. Com o método de quantização usado, para o *nbits* ≥ 3 começa a existir um nível de ruído significativo audível. Note-se que a conversão de float para inteiro (e vice-versa) e as redundâncias exploradas acabam, também, por contribuir ligeiramente para o aumento do ruído.

	sample01	sample02	sample03	sample04	sample05	sample06	sample07
nbits=1	4.45926	4.6812	3.80219	3.27126	2.95397	2.61309	6.42471
nbits=2	3.70181	3.96168	3.25203	2.99306	2.64895	2.21877	5.48423
nbits=3	3.21574	3.34716	2.82476	2.89281	2.38266	1.70532	4.62747
nbits=4	2.91721	2.78393	2.3889	2.78585	2.0414	1.20186	3.89618

Tabela 4. Entropia residual quantizada

Parte C

Exercício 1 - Relativamente à parte C, o objetivo é implementar um lossless codec de imagem baseado em predictive coding seguido de entropy coding através da classe Golomb já construída nos exercícios anteriores.

Antes do processo de codificação converteu-se a imagem para o formato YUV 4:2:0, utilizando a função **cvtColor** da biblioteca de openCV.

Após a conversão para YUV 4:2:0 foram obtidas as componentes Y, Cb e Cr da imagem, utilizando as seguintes fórmulas:

$$Y = 16 + 65.481R + 128.553G + 24.966B$$

$$Cb = 128 - 37.797R - 74.203G + 112.0B$$

$$Cr = 128 + 112.0R - 93.786G - 18.214B$$

em que :

R, G e B são as componentes de cada pixel

$$R, G, B \in [0, 1]$$

$$Y \in \{16, \dots, 235\}$$

$$Cb, Cr \in \{16, \dots, 240\}$$



Figura 2. Imagem original



Figura 3. Imagem YUV 4:2:0



Figura 4. Componente Y



Figuras 5 e 6. Componentes Cb e Cr

De forma a que as frames fossem codificadas através de predictive coding foi desenvolvido um non-linear predictor JPEG-LS, em que:

$$\hat{x} = \begin{cases} \min(a, b) & \text{if } c \geq \max(a, b) \\ \max(a, b) & \text{if } c \leq \min(a, b) \\ a + b + c & \text{otherwise} \end{cases}$$

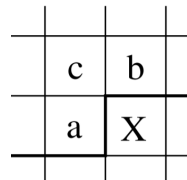


Figura 1. Esquema de pixels

Assim, foi criada a função **predictor** que cria os pixels a, b, c e \hat{x} , localizando-os de acordo com a **Figura 1** em que X representa o pixel atual. Foi também criada a função **pixelPredictor** que através dos pixels a, b e c criados é calculado \hat{x} usando o sistema em cima.

Uma vez que o entropy coding deve ser feito utilizando os códigos de Golomb, foi criado um codificador e um decodificador. Para o codificador, foram utilizados os resultados dos predictors das componentes Y, Cb e Cr, iterando-se e codificando-se cada pixel desses componentes utilizando as classes Golomb e bitStream. Para o decodificador, foram criadas 3 novas matrizes correspondentes a cada um dos componentes Y, Cb e Cr, iterou-se, e em cada pixel guardou-se o resultado do decodificador da classe Golomb no respetivo pixel da nova matriz.

Exercício 2 - O exercício 2 tem como objetivo utilizar o lossless codec implementado previamente e desenvolvê-lo de forma a permitir lossy coding. Para tal, foi criada a função **quantize** que cria uma nova imagem através da imagem original alterando cada pixel de forma a diminuir o número de bits do mesmo. A nova imagem criada é convertida para YUV 4:2:0 em vez da original.

Conclusão

Com este projeto foram desenvolvidos e consolidados conhecimentos acerca de áudio e imagem, bem como as bibliotecas utilizadas para tal.

Referências

- Documentation oficial de C++: <https://en.cppreference.com/w/>
- Biblioteca AudioFile: <https://github.com/adamstark/AudioFile>
- Biblioteca OpenCV: <https://opencv.org/>
- CMake: <https://cmake.org/>
- Binary Truncado: https://en.wikipedia.org/wiki/Truncated_binary_encoding