Entropy coding of audio and images

Universidade de Aveiro

Departamento de Eletrónica, Telecomunicações e Informática

Informação e Codificação

Relatório 2

António Domingues 89007 & Gabriel Saudade 89304 & Henrique Silva 88857

Professores: António Neves, Armando Pinho, Daniel canedo

27 de Dezembro de 2021

Introdução **3**

Parte A **4**
- **Exercício 1** **4**
- **Exercício 2** **5**
- **Exercício 3** **5**
- **Exercício 4** **6**

Parte B **7**
- **Exercício 3** **9**

Parte C **10**
- **Exercício 1** **10**
- **Exercício 2** **12**

Introdução

O presente relatório serve o propósito de explicar detalhadamente os métodos usados e raciocínios seguidos para a realização dos exercícios propostos no projeto 2 da cadeira de IC. Para cada exercício é feita uma descrição do código implementado e são apresentados os resultados obtidos.

O link para o repositório do github referente ao projeto é o seguinte:
https://github.com/antonioccdomingues/project02_IC

Parte A

Exercício 1

No exercício 1 da parte A foi criada uma classe **BitStream** para ler e escrever bits de e para ficheiros. A classe é composta por 4 métodos principais e para cada um será explicado o seu propósito e o modo como foi implementado.

O objetivo da função **string readnbits(string fnamein)** é ler os bits de um ficheiro, passado como parâmetro. Começa por ser instanciado um objeto do tipo **ifstream** para ler o ficheiro introduzido como parâmetro de entrada da função e é criada uma variável **bits** do tipo string onde são armazenados os bits lidos. De seguida é criado um ciclo **while** que prossegue enquanto houver informação a ser lida do ficheiro. Dentro deste ciclo while é criado um ciclo **for** de 7 a 0, para permitir obter cada bit de cada char lido do ficheiro. Dentro deste ciclo **for** cada bit do char é acedido, e para tal é feito um shift right de i posições no char e feito um **and** com 1 para determinar se o bit é 1 ou 0. De seguida os bits lidos vão sendo concatenados na string **bits**, que é retornada no fim da função.

A função **void read1bit(string fname, int nbyte, short nbit)** permite imprimir um bit de um byte, e tem como parâmetros de entrada o ficheiro a ler, o número do byte de onde se quer ler o bit e o índice do bit desejado. O procedimento seguido tem algumas diferenças relativamente ao da função **readnbits,** sendo inicializada uma variável **count** para além do objeto do tipo **ifstream**. O ciclo **while** procede enquanto houver informação a ser lida e enquanto a variável **count** não tiver chegado ao número do byte desejado. Dentro do ciclo, sempre que um char é lido do ficheiro o **count** é incrementado e quando tiver o valor do número do byte desejado, o valor do shift right feito no char lido é igual ao número do bit desejado. Por fim, o valor do bit desejado é impresso no terminal.

Quanto à função **void writenbits(string fnameout, string bitStr)** o objetivo passa por escrever num ficheiro passado como parâmetro de entrada, uma string de bits também passada como parâmetro. Primeiramente é criado um objeto do tipo **ofstream** para definir o ficheiro onde será escrita a informação. De seguida são criadas as variáveis **bittt**, **i, count** e **byteBuffer**. É implementado um ciclo while que permite percorrer todos os bits da string de bits e dentro do ciclo começamos por guardar o próximo bit da string na variável **bittt**. Se o valor do bit for 0, então é feito um **shift left** no **byteBuffer** para ficar lá armazenado um 0. Caso o valor do bit seja 1, então é feito um **shift left** no **byteBuffer** mas neste caso é ainda feito um **or** entre o bit e 1, para que o **byteBuffer** fique com um bit de valor 1 nesta posição. À medida que os bits vão sendo lidos, sempre que o índice do bit lido é múltiplo de 8, o conteúdo do **byteBuffer** é impresso no ficheiro de saída e é feito o seu reset, para que os bits impressos sejam separados em bytes. Caso o número de bits da string não seja múltiplo de 8 e já todos os bits tenham sido lidos, então as posições que faltarem para completar um número múltiplo de 8 bits, são preenchidas com 0's e o **byteBuffer** é impresso no ficheiro.

Por fim, a função **void write1bit(string fnameout, short bit)** permite escrever um bit num ficheiro de texto. Ambas as variáveis são passadas como parâmetro de entrada da função. Para o bom funcionamento desta função são criadas duas variáveis globais, **buff** e **cnt** que mantêm o valor sempre que a função é chamada. Dentro da função é instanciado um objeto do tipo **ofstream** para definir o ficheiro onde será escrito bit. Se o bit passado como parâmetro da função for 0, é inserido um 0 no **buff**. Se o bit passado for 1, então é inserido um

1 no **buff**. De seguida, a variável **cnt** é incrementada e sempre que o seu valor chegar a 8 o conteúdo do **buff** é impresso no ficheiro e é feito o reset de ambas as variáveis. Desta forma os bits são impressos no ficheiro de 8 em 8 chamadas da função, formando um byte.

Exercício 2

No seguimento do exercício 1 foi criado um programa que permite testar as funções implementadas e o seu bom funcionamento. Para tal foi criado o ficheiro **testEx1.cpp** onde é instanciado um objeto da classe **BitStream** e são chamadas as diferentes funções.

A primeira função a ser chamada é a **writenbits** com os seguintes parâmetros -> (bitfileout, "001111010010011000101010"). Desta forma é criado um ficheiro **bitfileout** cujo conteúdo são os caracteres **"=&*",** correspondentes aos bits passados como parâmetro e impressos no ficheiro.

De seguida é chamada a função **read1bit** com os parâmetros -> (bitfileout, 3, 1). Isto imprime no terminal o valor '1', pois o bit de índice 1 do terceiro byte do ficheiro **bitfileout** é 1.

A função **readnbits** é chamada com o parâmetro -> (bitfileout). A função retorna o conjunto de bits que compõem o ficheiro bitfileout, criado com a função **writenbits**. O conjunto é o seguinte 001111010010011000101010.

Por fim é chamada a função **write1bit.** Esta função é chamada 8 vezes de modo a imprimir no ficheiro **bitfileout2** os seguintes bits -> 00100011 (bit a bit). Este conjunto de bits corresponde ao caractere **"#",** que é o resultado da impressão no ficheiro **bitfileout2.**

Exercício 3

Neste exercício, é pedido que seja criada uma classe cuja função seja codificar um número, usando os códigos de **golomb**. Numa primeira abordagem, foi estudado o algoritmo que se encontrava nos apontamentos da teórica. De seguida percebeu-se que este código era composto por duas partes, a esquerda pelo código unário, e a direita com código binário, mas com algumas restrições. Para além disto, uma outra dependência, um parâmetro *m,* que caso seja potência de 2, aplica o algoritmo de golomb de uma certa maneira, e caso o *m* não seja potência de 2, aplica de outra.

Após estudar este algoritmo, aplicou-se o mesmo, como se encontra na pasta include/Golomb.h.

Nos exercícios das partes seguintes, é necessário codificar as samples usando este código. Para tal, é necessário calcular o *m* ótimo, para o conjunto de valores a codificar, para que a codificação seja o mais compacta possível. Como vimos na teoria, o código golomb é ótimo para um *m,* que segue uma distribuição geométrica com média (1-p)/p.

Foi implementada outra função (***decoderByBit***), cuja função é decodificar os valores provenientes da função ***readnbits***. Esta recebe uma string, com todas as palavras descodificadas, assim como o valor m para descodificar as mesmas. Depois é feita uma análise sequencial, adicionando a um *vector,* as palavras que vão sendo validadas. No fim retorna um vector, com todos os inteiros descodificados. Foram realizados os respetivos testes,

que vão ser explicados de seguida. Foi adicionado um ficheiro *readme, para informar o utilizador, do processo de execução dos testes.*

Exercício 4

Neste exercício, é posto à prova a classe ***Golomb.h***, implementada anteriormente. Na primeira fase dos testes, apenas são codificados 2 valores (que se encontram nos slides), e esses mesmos são descodificados. Com isto é suposto a descodificação dar o mesmo que o valor que era suposto ser codificado.

De seguida, é criado um segundo conjunto de testes mais exaustivos, que permitem ao utilizador selecionar a quantidade de valores a testar entre 0 e 50. É calculada a média destes valores para obtenção do valor ótimo de *m.* Depois é calculado um valor p, e de seguida o m, de acordo com a fórmula da teórica.

```
for(int i = 0; i< valores; i++){     // cálculo do m ótimo
        soma += valor[i];
    }
    media = soma/valores;
    double p = 1/(media+1);
    int m = ceil(-(1/(log2(1-p)))); //obtenção do valor m
```

Posteriormente, são codificados esses valores e guardados num array de strings, para serem descodificados no próximo passo. É feito um assert, para garantir que o valor descodificado, é o mesmo que estava a ser codificado.

Pode-se assim concluir, que estas duas funções estão a funcionar corretamente, e que estão síncronas uma com a outra, pelo que os valores codificados e descodificados estão em sintonia.

Parte B

Exercício 1

O objetivo do exercício 1 passa por implementar um codec de áudio usando a classe de Golomb desenvolvida no exercício anterior, com o intuito de obter a maior taxa de compressão possível. Desta forma foi implementada uma class **Preditor**.

Inicialmente são extraídas e convertidas todas as samples de cada canal do do ficheiro de áudio. Como a biblioteca usada para extrair as samples foi a **AudioFile.h**, um dos problemas que surgiu foi a conversão do tipo de dados de floating para inteiro. Para alcançar a conversão do tipo de dados, a cada sample foi multiplicado por um fator correspondente a 2^15 (valor mínimo de alcance de um valor inteiro na linguagem C).

De seguida começamos por realizar a previsão dos valores. A técnica usada para a previsão do valor das samples foi um polinômio de previsão que usa as últimas 3 samples com o objetivo de minimizar o erro de previsão e maximizar a precisão do valor previsto, tal que *xn* corresponde ao valor da sample n.

$$previsto \ = \ 3x_{n-1} \ - \ 3x_{n-2} \ + \ x_{n-3}$$

Calculado o valor previsto, é deduzido o valor residual tal que:

$$rn \ = xn_{n} - previsto[n \ - \ 3]$$

onde *rn* é o valor residual obtido através da subtração do valor real da sample com o valor previsto.

Após a obtenção dos valores residuais, é usada a codificação de Golomb para a compressão das samples de áudio, de tal maneira que é calculado o parâmetro de Golomb-Rice M e a cada valor residual codificado é acrescentado um bit à direita (bit menos significativo) que corresponde ao sinal da sample.

Posto isto, é possível comparar a entropia do ficheiro inicial de áudio com a entropia prevista (correspondente ao valor residual). Na tabela abaixo é possível observar como a previsão afetou e minimizou a entropia do ficheiro de áudio na sua compressão.

	Inicial Entropy	**Obtained Entropy**	**Compression Time**
sample01.wav	12192,25	3115,73	4.227
sample02.wav	6668,07	3212,47	2.332
sample03.wav	7980,755	1555,715	2.394
sample04.wav	12590,5	961,0605	1.483
sample05.wav	7070,46	608,085	2.128
sample06.wav	4941,405	105,844	2.195
sample07.wav	23812,45	8981,935	5.791

Observando os resultados obtidos, o tempo de compressão é aceitável e a entropia do ficheiro áudio obtido é menor. Seria de esperar que o tamanho do ficheiro obtido fosse menor, no entanto a razão da compressão aumentou.

Na descompressão do ficheiro de áudio foi usado o método inverso de previsão. Através dos valores residuais lidos do ficheiro binário é obtido o valor previsto e somado ao valor residual tal como é possível observar nas equações abaixo.

$$previsto = 3rn_{n-1} - 3rn_{n-2} + rn_{n-3}$$

$$xn = previsto_{n} + rn$$

Visto que a leitura do ficheiro não corresponde aos bits inseridos inicialmente, foi impossível obter o ficheiro de áudio original. No entanto, é possível obter o tempo de descompressão usando a codificação de Golomb para as samples 1 e 2.

	Decompression Time
sample01.wav	6.554
sample02.wav	4.202

Exercício 3

De modo a obter o maior grau de compressão possível em tempo útil, foi implementado um codec baseado em quantização residual. Tendo em conta o que foi implementado durante o projeto 1, foi adicionado ruído branco a cada sample de modo a simular a distorção do áudio no ficheiro. A operação de quantização é penosa no tempo de execução do programa e portanto não é praticada do ponto de vista da compressão. No entanto, os tempos de compressão diminuem em média 0.5 segundos, como é possível observar na tabela abaixo.

	Compression Time
sample01.wav	3.649
sample02.wav	1.781
sample03.wav	2.029
sample04.wav	1.277
sample05.wav	2.064
sample06.wav	2.204
sample07.wav	3.864

Parte C

Exercício 1

Nesta parte, vai ser analisado o processo de resolução do exercício, ou seja, quais são os passos necessários para fazer a codificação da imagem, assim como a descodificação da mesma. Vai também ser feita uma análise de resultados. Por último, uma breve descrição do porquê destes resultados, assim como os obstáculos encontrados.

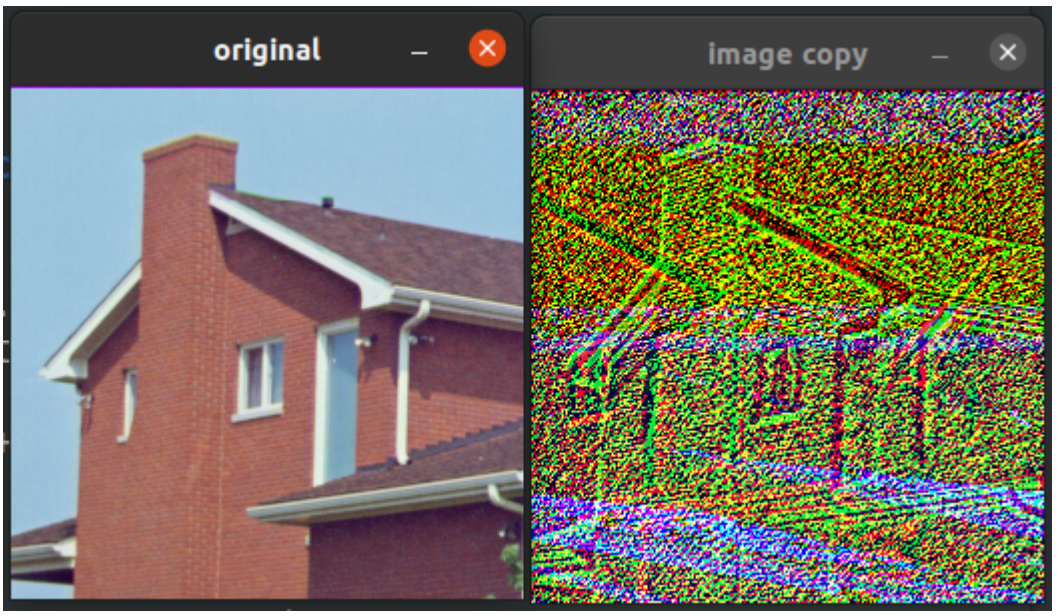
Numa primeira fase, a imagem necessita de ser convertida do formato *RGB*, para o formato *YUV*. Para a obtenção de uma imagem ***lossless***, é necessário fazer a prediction de cada pixel da imagem original, através do ***nonlinear predictor of JPEG-LS***, fornecido nas aulas teóricas. Mas isto não é suficiente para codificar uma imagem com tamanho mais compacto. De seguida são calculados os valores residuais, ***valores dos pixels originais - valores da prediction.*** Os valores a codificar, são os valores residuais. Estes são codificados usando os códigos de ***Golomb***, utilizando a classe BitStream, mais concretamente a **writenbits**, para escrever os valores codificados, num ficheiro binário. Um dos nossos objetivos, foi o cálculo do valor ótimo de m (como explicado anteriormente) antes da codificação dos valores, para se obter uma codificação e descodificação bastante razoável.

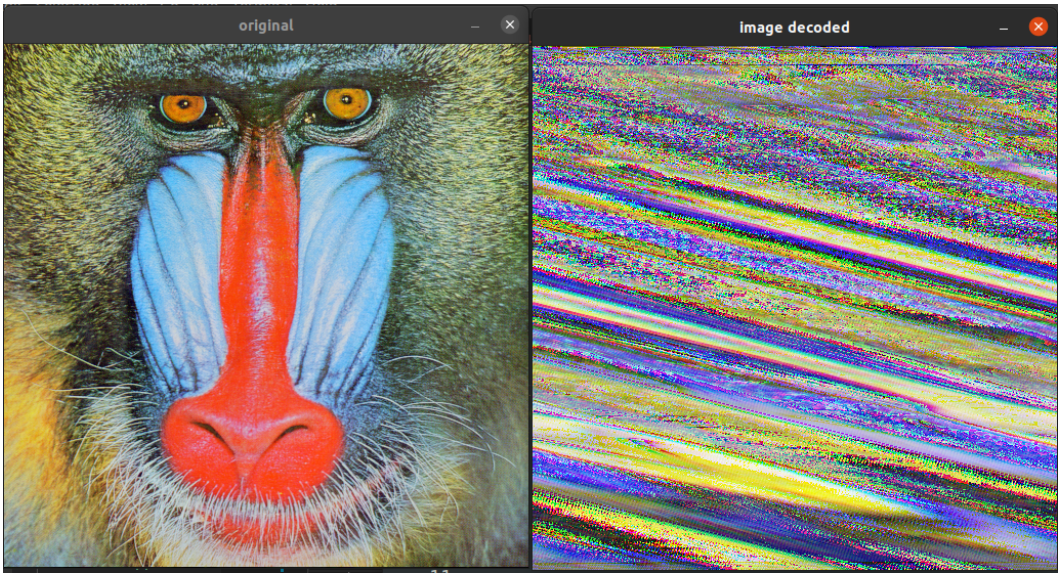
O descodificador, por sua vez, vai utilizar a função readnbits, lendo todos os bits do ficheiro(retornando uma string) onde outrora foram codificados os valores residuais. Esta string, é passada à função ***decoderByBit***, que por sua vez retorna um ***vector***, com todos os inteiros descodificados (tendo também em conta o sinal). Ao estar neste passo da descodificação, apenas basta fazer o processo inverso da codificação, ou seja, fazer a prediction dos valores descodificados (residual), somar essa prediction com os valores descodificados, obtendo assim os valores originais da imagem, mas do lado do descodificador. De seguida, apenas é feito o display da imagem descodificada, e assim se obtém uma imagem ***lossless***.

Relativamente a resultados obtidos, os mesmos não são os esperados. Em algumas das imagens fornecidas, a imagem descodificada, tem mínimos traços semelhantes em comparação com a imagem original. Isto acontece pelo facto da **writenbits** escrever determinados valores, e a **readnbits** obter valores distintos dos quais foram escritos. Abaixo, temos 2 exemplos de resultados finais. Não são apresentados mais exemplos, pois são todos semelhantes a esses abaixo apresentados.

Como referido muito brevemente acima, os valores escritos no ficheiro binário através da função writenbits, não são iguais aos lidos na função reanbits. Só por aí, a imagem descodificada, nunca vai ser igual à original, visto que os valores nem sequer são os mesmos. Quanto a tempos de codificação, estes são praticamente instantâneos. Na descodificação, esta já apresenta tempos mais demorados. Isto acontece quando estamos a descodificar valores de ***m***'s não potência de 2. Deve-se pelo facto que essa função (***decoderByBit***) apresenta nível de complexidade ***n^2***, pois tem um *for* dentro de outro *for* (apenas quando m= **não** potência de 2).

Para concluir, a lógica do codificador e descodificador encontra-se toda implementada. Como trabalho futuro, resolvendo esses 2 contratempos, o programa deve mostrar a imagem descodificada igual à original.

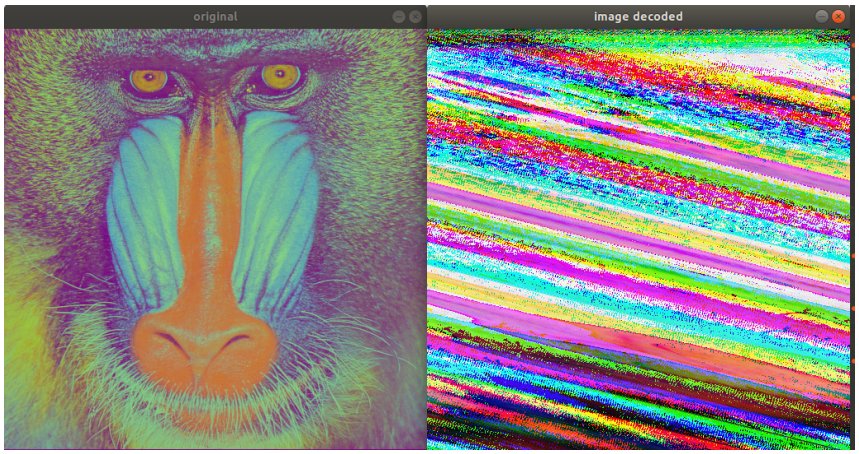

original
image copy


original
image decoded

Exercício 2

No exercício 2 o procedimento seguido é bastante semelhante ao do exercício 1, com a diferença de que neste caso cada byte de cada pixel da imagem inicial sofre o shift de bits correspondente ao valor de quantização inserido pelo utilizador. O valor do byte é shiftado à esquerda para garantir que não são perdidos bits e depois é shiftado à direita para garantir que o valor do byte é reduzido. Desta forma o valor dos bytes usados diminui e o peso total da imagem é reduzido. Depois disso procede-se à codificação da imagem e posterior descodificação.

Os resultados obtidos são apresentados de seguida.



A imagem original perde alguma cor devido aos deslocamentos feitos nos bytes de cada pixel que reduzem a vivacidade das cores disponíveis. Na imagem descodificada as razões mais prováveis para o resultado não ser o desejado são as expostas no exercício 1.