Compressão de Entropia

1 Padrão CUIF

Na última aula vimos um exemplo simples de um padrão simples de representação de imagens, o chamado padrão CUIF Versão 1 (CUIF.1). Segue abaixo o formato do cabeçalho CUIF, como já visto na aula prática anterior.

Offset	Tamanho	Descrição
0	2	assinatura (identificador), deve ser 5431_{10}
2	2	versão do padrão `CUI`
3	1	número de estudantes no grupo (NUMBER_OF_STUDENTS)
4	4	largura da imagem (em pixels)
8	4	altura da imagem (em pixels)

Após o cabeçalho, há uma lista de números de matrículas dos alunos que formam o grupo da aula prática. Cada matrícula ocupará 4 bytes. Note que o número de alunos do grupo deve estar definido corretamente no cabeçalho (NUMBER_OF_STUDENTS).

Após 12 bytes do cabeçalho + 4× NUMBER_OF_STUDENTS bytes, estarão os dados da imagem.

2 Baixando projeto para a aula prática

Nesta aula prática, veremos mais 3 versões do padrão CUIF. Então ao todo teremos 4 versões:

- CUIF.1. Representa a imagem, sem compressão, em um formato RGB;
- CUIF.2. Representa a imagem, sem compressão, em um formato YCbCr.
- CUIF.3. Representa a imagem em um formato YCbCr, com compressão usando codificação de Huffman.
- CUIF.4. Representa a imagem em um formato RGB, com compressão usando codificação RLE.

Para esta aula prática, baixe o projeto Java no Moodle. Este projeto contêm 3 pacotes:

- huffmancoding: que implementa a codificação de Huffman. Não há necessidade de se estudar este pacote, bastando entender apenas os métodos br.ufsc.ine5431.huffmancoding.HuffamnCoding.compression e br.ufsc.ine5431.huffmancoding.HuffamnCoding.decompression que fazem a compactação e descompactação usando a codificação de Huffman;
- rlecoding: que contém a classe RLECoding, com seus métodos para compactação e descompactação usando a codificação RLE;
- praticaiv: pacote contendo as classes usadas nas aulas práticas.

Lembrando que usamos duas ferramentas para conversão:

1. `bmp2cuif`: para conversão de BMP para CUIF. A sintaxe deste comando é *java bmp2cuif -v 1 input.bmp output.cuif*, onde a versão pode varia de 1 a 4;
2. `cuif2bmp`: para conversão de CUIF para BMP. A sintaxe deste comando é *java cuif2bmp input.cuif output.bmp* ;

3 CUIF.1

Como visto na aula prática anterior, o padrão `CUI.1` é uma representação RGB separada em canais, de maneira similar ao BMP. Porém, diferente do BMP onde cada *pixel* aparece com seus canais BGR, o `CUI.1` apresenta cada canal R, G e B completos em sequência. Ou seja, ao invés de codificar *pixel*-a-*pixel*, codifica-se canal-a-canal. Cada *pixel* utilizará 1 byte em cada canal.

Exemplo de `CUI.1`, representado em um arquivo CUIF

Vamos supor uma imagem com 2×2 *pixels*:

red = $(\text{FF } 00\ 00)_{16}$ green = $(00\ \text{FF } 00)_{16}$
blue = $(00\ 00\ \text{FF})_{16}$ gray = $(\text{B7 B7 B7})_{16}$

Para este exemplo, há apenas um estudante no grupo, cuja matrícula é 99132042. Vejamos como fica a organização de um arquivo CUIF para armazenar essa imagem seguindo o padrão `CUI.1`:

byte	valor				significado
	3	2	1	0	
0	–	–	5431_{10}		assinatura CUIF
2	–	–	–	1	versão do padrão CUI (CUI.1)
3	–	–	–	1	número de estudantes no grupo
4	2_{10}				largura
8	2_{10}				altura
12	99132042_{10}				matrícula do aluno no grupo
16	–	–	–	FF_{16}	R pixel 0,0
17	–	–	–	00_{16}	R pixel 0,1
18	–	–	–	00_{16}	R pixel 1,0
19	–	–	–	$B7_{16}$	R pixel 1,1
20	–	–	–	00_{16}	G pixel 0,0
21	–	–	–	FF_{16}	G pixel 0,1
22	–	–	–	00_{16}	G pixel 1,0
23	–	–	–	$B7_{16}$	G pixel 1,1
24	–	–	–	00_{16}	B pixel 0,0
25	–	–	–	00_{16}	B pixel 0,1
26	–	–	–	FF_{16}	B pixel 1,0
27	–	–	–	$B7_{16}$	B pixel 1,1

4 Roteiro da Parte I

Faça os seguintes procedimentos inciais:

1. Baixar o projeto `CUI` no Moodle.
2. Modificar o valor da variável `numero_de_estudantes` (linha 47 do arquivo `bmp2cuif.java`) para o número de estudantes no grupo;
3. Atualizar o array (`id_estudantes`, linha 48, `bmp2cuif.java`) com seus números de matrícula dos alunos do grupo de alunos;
4. Gerar os arquivos JazzMan2.cuif e JazzMan2.bmp usando:

   ```
   java bmp2cuif -v 1 JazzMan.bmp JazzMan1.cuif
   ```

   ```
   java cuif2bmp JazzMan1.cuif JazzMan1.bmp
   ```

5. Corrigir o código projeto caso necessário (Caso não tenha feito, volte para a prática III).

5 CUIF.2

Nosso padrão `CUI.2` é bastante similar ao `CUI.1`. Porém, ao invés de representar os *pixels* em RGB, a representação será em YCbCr, de acordo com a recomendação ITU-R BT.601. O documento descrevendo tal recomendação está disponível em https://www.itu.int/dms_pubrec/itu-r/rec/bt/R-REC-BT.601-7-201103-I!!PDF-E.pdf.

Segue na sequência o formato do arquivo CUIF.2

Exemplo de `CUI.1`, representado em um arquivo CUIF

Vamos supor uma imagem com 2×2 *pixels*. Para este exemplo, há apenas um estudante no grupo, cuja matrícula é 99132042. Vejamos como fica a organização de um arquivo CUIF para armazenar essa imagem seguindo o padrão `CUI.1`:

byte	valor				significado
	3	2	1	0	
0	–	–	5431_{10}		assinatura CUIF
2	–	–	–	2	versão do padrão CUI (CUI.1)
3	–	–	–	1	número de estudantes no grupo
4	2_{10}				largura
8	2_{10}				altura
12	99132042_{10}				matrícula do aluno no grupo
16	–	–	–	FF_{16}	Y pixel 0,0
17	–	–	–	00_{16}	Y pixel 0,1
18	–	–	–	00_{16}	Y pixel 1,0
19	–	–	–	$B7_{16}$	Y pixel 1,1
20	–	–	–	00_{16}	Cb pixel 0,0
21	–	–	–	FF_{16}	Cb pixel 0,1
22	–	–	–	00_{16}	Cb pixel 1,0
23	–	–	–	$B7_{16}$	Cb pixel 1,1
24	–	–	–	00_{16}	Cr pixel 0,0
25	–	–	–	00_{16}	Cr pixel 0,1
26	–	–	–	FF_{16}	Cr pixel 1,0
27	–	–	–	$B7_{16}$	Cr pixel 1,1

6 Roteiro da Parte II

Faça os seguintes procedimentos com o CUIF.2:

1. Baixar a recomendação BT.601, ler as Seções 2.5.1 e 2.5.2 e implementar a conversão RGB → YCbCr no método `rgb_to_ycbcr_base_double` do arquivo `ColorSpace.java` (linha 15);
2. Gerar um arquivo chamado `JazzMan2.cuif` usando:

   ```
   java bmp2cuif -v 2 JazzMan.bmp JazzMan2.cuif
   ```

3. Gerar um arquivo chamado `JazzMan2.bmp` usando o comando a seguir e abra o arquivo bmp. Caso haja algum problema na imagem, é necessário verificar a conversão RGB para YCbCr implementada.

   ```
   java cuif2bmp JazzMan2.cuif JazzMan2.bmp
   ```

7 Calculando Ruído

A figura abaixo representa um esquema de codificação e decodificação. Na entrada, a mensagem original (Ori) é codificada e posteriormente decodificada, resultando em uma mensagem decodificada (Dec). Em uma codificação com perdas de informações, a mensagem decodificada (Dec) é diferente da mensagem original (Ori). Neste caso, diz-se que o codificador gerou ruído na mensagem original.

Há diversas formas para medir o erro gerado pelo codificador. Uma destas métricas é a Média dos Erros Quadráticos (MSE – *Mean Squared Error*). Considerando que tanto Ori quanto Dec tenham tamanho n, cada símbolo pode ser indexado, respectivamente, como ori_i e dec_i, onde $1 \leq i \leq n$. Vamos assumir que os símbolos sejam valores numéricos, assim, a MSE é definida como:

$$\text{MSE}(\text{Ori}, \text{Dec}) = \frac{1}{n}\sum_{i=1}^{n}(\text{ori}_i - \text{dec}_i)^2 \quad (1)$$

Para determinar o MSE entre uma imagem Ori e uma imagem Dec, o tamanho de símbolos é a quantidade de componentes de cor de uma imagem (3 componentes por píxel). Assim, o tamanho n de uma imagem de largura w e altura h é igual a $3*h*w$.

Outra métrica bastante utilizada, principalmente na compressão de sinais é a chamada Relação Sinal-Ruído de Pico (PSNR – *Peak Signal-to-Noise Ratio*), definida (em dB) como:

$$\text{PSNR}(\text{Ori}, \text{Dec}) = 10 \times log_{10}\left(\frac{(2^b - 1)^2}{\text{MSE}(\text{Ori}, \text{Dec})}\right) \quad (2)$$

Onde b é o número de bits por símbolo. Quanto maior o PSNR melhor é a qualidade da imagem, ou seja, menores são os erros. Notem que quando Ori e Dec são iguais, isto é, quando não há erros, PSNR = ∞. Utilizaremos a PSNR para medir os erros causados em nossos padrões `CUI`.

8 Roteiro da Parte III

1. Analise a classe *PSNR.java* que calcula o PSNR a partir de um BMP original e um BMP decodificado a partir de um arquivo CUIF.x.
2. Implemente os métodos *mae* e *psnr* da classe *PSNR.java* (linha 203) para retornar o PSNR obtida pela conversão RGB para YCbCr.
3. Teste a implementação usando o comando a seguir, sendo que JazzMan1.bmp foi decodificado a partir do CUIF.1. Como os píxeis são iguais, o valor do PSNR deveria ser infinito.

   ```
   java PSNR JazzMan.bmp JazzMan1.bmp
   ```

9 Questões a serem respondidas no relatório

Entregue no relatório o código fonte com as implementações realizadas e com as respostas das seguintes questões:

Questão 1. Há alguma compressão entre `CUI.1` e `CUI.2`? Expliquem.

Questão 2. Indique o PSNR comparando a imagem original com as imagens obtidas a partir dos arquivos CUIF.1 e CUIF2. Há perdas nos dados da imagem na conversão RGB → YCbCr → RGB? Explique porque.

10 Cuif.3 Compactado usando Codificação de Huffman

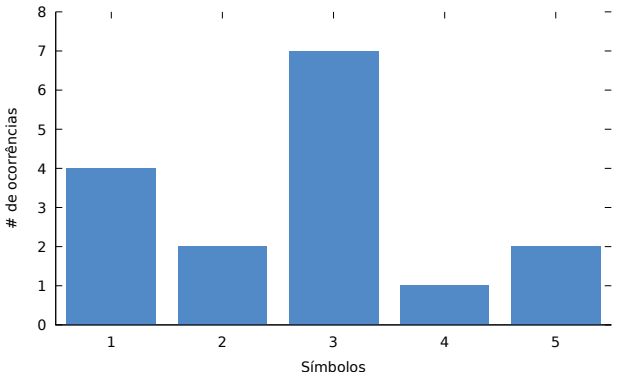
Vimos na aula teórica como funciona a codificação de Huffman para uma mensagem cujas distribuições de probabilidade são conhecidas (ou são calculadas). Devemos lembrar que a codificação de Huffman codifica símbolos que ocorrem mais com menos bits, e codifica símbolos que ocorrem menos frequentemente com mais bits. Para isso, é necessário primeiro construir uma linha de probabilidade acumulada de ocorrência dos símbolos.

A nossa implementação de Huffman utiliza histogramas para obter as frequências de ocorrências dos símbolos usadas na criação das árvores de Huffman. Basicamente, para cada símbolo há uma contagem de sua ocorrência. Ou seja, um histograma é um mapeamento $histogram(x)$ entre o símbolo x e o número de ocorrências de x em uma mensagem m. Por exemplo, supondo a mensagem $m = \{3, 3, 2, 5, 1, 1, 1, 2, 3, 5, 1, 4, 3, 3, 3, 3\}$, $histogram(.)$ pode ser apresentado na forma da Tabela 1. Outra forma de visualizar histogramas é através de um gráfico, como exemplificado na Figura 1(a), onde o eixo das abscissas apresenta os símbolos, enquanto o eixo das ordenadas apresenta a frequência de ocorrência. A Figura 2(b) apresenta o histograma da imagem lena.bmp.

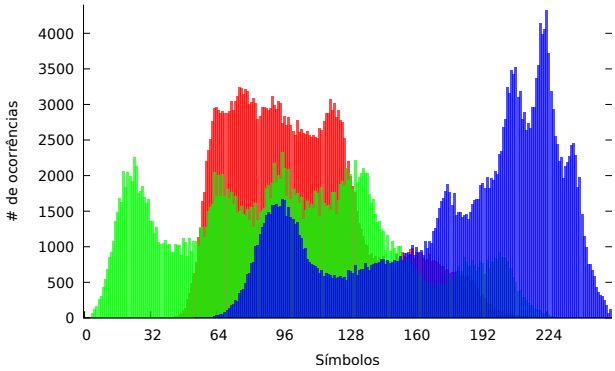
Na implementação disponibilizada, o CUIF.3 converte a representação RGB para YCbCr antes de executar a codificação de Huffmann. Note que a codificação de Huffman em si não gera descarte de informação, por ser uma técnica de codificação sem perdas.

Tabela 1: Exemplo de histograma

símbolo	# de ocorrências
1	4
2	2
3	7
4	1
5	2
total	16



(a) Exemplo de histograma de m.



(b) Exemplo de histograma dos canais R, G e B da imagem `lena.bmp`.

Figura 1: Exemplos de histogramas.

11 Roteiro da Parte IV

1. Execute os comandos a seguir para gerar arquivo JazzMan.cuif na versão 3 e decodificar esta imagem novamente para bitmap. Observe o console, onse será apresentado o histograma da imagem e a tabela de codificação de Huffman. Note o princípio da codificação de Huffman, onde os valores que ocorrem mais são codificados com menos bits.

   ```
   java bmp2cuif -v 3 JazzMan.bmp JazzMan3.cuif
   ```

   ```
   java cuif2bmp JazzMan3.cuif JazzMan3.bmp
   ```

2. Obtenha o Erro PSNR do arquivo CUIF.3 decodificado, comparando com o CUIF.1 e CUIF.2

   ```
   java PSNR JazzMan.bmp JazzMan3.bmp
   ```

   ```
   java PSNR JazzMan2.bmp JazzMan3.bmp
   ```

12 Questões a serem respondidas no relatório

Entregue no relatório as respostas das seguintes questões:

Questão 3. Calcule a taxa de compressão em relação ao arquivo JazzMan1.cuif (CUIF.1 sem compressão) e JazzMan3.cuif (CUIF.3 com Codificação de Huffman). Dica: A taxa de compressão da versão V em relação à versão X é $\frac{\text{tamanho CUI.X}}{\text{tamanho CUI.V}} : 1$. Por exemplo, se CUI.X resulta em 100 KB e CUI.V resulta em 50 KB, a taxa de compressão de V em relação a X é de 2:1. Comente também no relatório o princípio da compressão usada na codificação de Huffman, para isto analise o histograma e a tabela de códigos impressos.

Questão 4. Indique o PSNR comparando a imagem original JazzMan.bmp com a imagem obtida a partir do arquivo CUIF.3. Há perdas nos dados da imagem? Explique porque.

13 Cuif.4 Compactado usando RLE

Na implementação de RLE disponibilizada, a compressão se baseia na supressão de repetição de símbolos de tamanho de 1 byte (8bits). Nesta implementação, o bit mais significativo de cada byte foi utilizado como flag para indicar se o byte representa um símbolo único (sem repetição) ou contém o número de repetições do próximo byte. Assim, caso o bit mais significativo for 0, o byte representa um símbolo único. Caso este bit seja 1, o restante dos bits indica o número de repetição, e o próximo byte é o símbolo que se repete.

Para melhor compreender como funciona o RLE em nossa implementação, vamos analisar a mesma mensagem que abordamos anteriormente: $m = \{3, 3, 2, 5, 1, 1, 1, 2, 3, 5, 1, 4, 3, 3, 3, 3\}$. Esta, seria codificada em binário (1 byte por símbolo) como:

Tabela 2: Codificação binária de m considerando 8 bits/símbolo.

símbolo	binário
3	00000011
3	00000011
2	00000010
5	00000101
1	00000001
1	00000001
1	00000001
2	00000010
3	00000011
5	00000101
1	00000001
4	00000100
3	00000011
3	00000011
3	00000011
3	00000011

Já na codificação RLE proposta na implementação, a representação seria da forma a seguir:

Tabela 3: Codificação RLE conforme implementada para `CUI.4`

binário	significado
10000010	o próximo byte é repetido 2× (3, 3)
00000011	byte que é repetido: 3
00000001	este byte é único, representa 2
00000010	este byte é único, representa o 4 (deveria ser 5)
10000011	o próximo byte é repetido 3× (1, 1, 1)
00000001	este byte é repetido: 1 (não há erro)
00000001	este byte é único, representa o 2
00000001	este byte é único, representa o 2 (deveria ser 3)
00000010	este byte é único, representa o 4 (deveria ser 5)
00000000	este byte é único, representa o 0 (deveria ser 1)
00000010	este byte é único, representa o 4
10000100	o próximo byte é repetido 4× (3, 3, 3, 3)
00000011	byte que é repetido: 3

Esta implementação suporta apenas entradas de até 127 símbolos, por utilizar 7 bits para codificar os símbolos (primeiro bit é usado como flag de repetição). Como as imagens consideradas representam valores de Y, Cb e CR com 1 byte (valores de 0 a 255), o valor do byte é dividido por 2 na codificação e multiplicado por 2 na decodificação. Na implementação, foi utilizado o deslocamento de um bit a direita na codificação e deslocamento de um bit a esquerda na decodificação. Assim, sempre que o byte representando um símbolo único for ímpar, haverá erro no bit menos significativo, que será zerado.

A mensagem decodificada seria: $m' = \{3, 3, 2, 4, 1, 1, 1, 2, 2, 4, 0, 4, 3, 3, 3, 3\}$. Note que há erros! Usando a Equação 1 podemos estimar a MSE entre m e m':

$$MSE(m, m') = \frac{1}{16} \times (1^2 + 1^2 + 1^2 + 1^2) = \frac{1}{16} \times 4 = \frac{1}{4}$$

Através da MSE, podemos obter a PSNR (Eq. 2):

$$PSNR(m, m') = 10 \times log_{10} \left(\frac{(2^8 - 1)^2}{MSE(m, m')} \right) = 10 \times log_{10} \left(\frac{65025}{1/4} \right) = 10 \times log_{10}(260100) = 54,1514dB$$

Vimos então que, mesmo a RLE sendo um método de codificação de entropia que por definição não gera erros, por conta de decisões de implementação foram inseridos erros. Durante a execução do roteiro a seguir, busquem analisar as imagens codificadas e verifiquem se é possível identificar esses erros.

14 Roteiro da Parte V

1. Usando o bmp2cuif, gere, a partir da JazzMan.bmp arquivo no formato CUIF.4, chamado JazzMan4.cuif e em seguida gere o arquivo bmp a partir de cada um dos arquivos cuif (chamado JazzMan4.bmp).
2. Calcule a PSNR das codificações CUIF.4, usando para tal o arquivo JazzMan.bmp (original) e o arquivo JazzMan4.bmp.

15 Questões a serem respondidas no relatório

Entregue no relatório as respostas das seguintes questões:

Questão 5. Qual a taxa de compressão obtida com CUIF.4?).

Questão 6. Indique a PSNR das codificações CUIF.4. Compare com a PSNR do CUIF3 e justifique.

Questão 7. Codifiquem as imagens JazzMan.bmp e lena.bmp usando CUIF.4. Qual imagem obteve maior compressão? Explique porque.