图像处理与机器视觉

# 第三次作业

## 题目一

对给定图像lena.bmp进行霍夫曼编码。在word中贴出代码，以及编码的码字结果、编码的平均长度、压缩比值。

### 题目分析

霍夫曼编码的编码过程如下：

1. 将信源符号按出现概率从大到小排成一列，然后把最末两个符号的概率相加，合成一个概率。重复上述做法，直到最后剩下两个概率为止。

2. 从最后一步剩下的两个概率开始逐步向前进行编码。每步只需对两个分支各赋予一个二进制码，如对概率大的赋予码 0，对概率小的赋予码 1（反之亦可）。

根据以上步骤，我们需要先读取和计算图像中的灰度值的概率，接下来根据概率构造霍夫曼树（即步骤1）。完成构建后，通过遍历这棵霍夫曼树得到霍夫曼编码。编码的平均长度由概率乘以对应编码长度求和算得，压缩比值等于8除以平均长度。

### 代码实现

clear;

clc;

img = imread('lena.bmp');

*% 计算灰度图像中每个像素值的频率*

numPixels = numel(img);

[uniqueSymbols, ~, symbolCounts] = unique(img);

symbolCounts = accumarray(symbolCounts, 1);

symbolProbabilities = symbolCounts / numPixels;

*% 构建霍夫曼树和霍夫曼编码*

huffmanTree = buildHuffmanTree(uniqueSymbols, symbolProbabilities);

huffmanCodes = generateHuffmanCodes(huffmanTree, uniqueSymbols);

*% 这是因为huffmanCodes实际上是symbol和code构成的二维数组，且以code为顺序升序*

huffmanCodes = sortrows(huffmanCodes, 1);

fprintf('Symbol Probability Huffman Code\n');

**for** i = 1:numel(uniqueSymbols)

fprintf('%d\t\t%f\t\t%s\n', huffmanCodes{i,1},symbolProbabilities(i), huffmanCodes{i,2});

**end**

*% 平均长度*

averageLength = sum(symbolProbabilities .\* cellfun(@length, huffmanCodes));

fprintf('Average Length: %f\n',averageLength(2));

*% 压缩比值*

compressionRatio = 8 / averageLength(2);

fprintf('Compression Ratio:%f\n ',compressionRatio);

*% 构建霍夫曼树*

**function** **huffmanTree** = **buildHuffmanTree**(symbols, probabilities)

numSymbols = numel(symbols);

leafNodes = cell(numSymbols, 1);

**for** i = 1:numSymbols

leafNodes{i} = struct('symbol', symbols(i), 'probability', probabilities(i));

**end**

**while** numel(leafNodes) > 1

*% 按概率升序排序叶子节点*

[~, sortedIndices] = sort(cellfun(@(x) x.probability, leafNodes));

sortedLeafNodes = leafNodes(sortedIndices);

*% 取最小概率的两个节点，合并为一个新节点*

newNode = struct('left', sortedLeafNodes{1}, 'right', sortedLeafNodes{2}, 'symbol', [],...

'probability', sortedLeafNodes{1}.probability + sortedLeafNodes{2}.probability);

*% 从叶子节点列表中移除被合并的节点*

leafNodes = sortedLeafNodes(3:**end**);

*% 将新节点添加到叶子节点列表*

leafNodes{end+1} = newNode;

**end**

huffmanTree = leafNodes{1};

**end**

*% 生成霍夫曼编码*

**function** **codes** = **generateHuffmanCodes**(tree, symbols, prefix)

**if** nargin < 3

prefix = '';

**end**

**if** ~isempty(tree.symbol) *% 叶子*

index = find(strcmp(symbols, tree.symbol)); *% 找到符号在列表中的索引*

codes = {tree.symbol, prefix}; *% 以符号为键值对应编码*

**else** *% 非叶子*

leftCodes = generateHuffmanCodes(tree.left, symbols, [prefix '0']);

rightCodes = generateHuffmanCodes(tree.right, symbols, [prefix '1']);

codes = [leftCodes; rightCodes];

**end**

**end**

### 结果展示

代码输出结果如下：

Symbol Probability Huffman Code

10 0.000015 1110110011001010

11 0.000015 1110110011001011

12 0.000046 00110100000101

14 0.000031 001101000010101

15 0.000031 001101000010110

16 0.000046 00110100000110

17 0.000168 001101000000

18 0.000244 100010101010

19 0.000427 01001001110

20 0.000809 0100100110

21 0.000687 11101100111

22 0.001144 1100111100

23 0.001358 000000110

24 0.001694 010010010

25 0.002472 111000011

26 0.003281 00111111

27 0.003784 01101100

28 0.005264 11111000

29 0.005798 0010000

30 0.007599 0110111

31 0.009079 1100010

32 0.009872 1110100

33 0.010254 1111001

34 0.010529 1111011

35 0.011124 000010

36 0.009857 1110011

37 0.009430 1101110

38 0.009216 1101001

39 0.008118 1000111

40 0.007538 0110011

41 0.006210 0011011

42 0.005035 11101111

43 0.005463 0000000

44 0.005280 11111010

45 0.004623 11011010

46 0.004089 10010011

47 0.004028 10000110

48 0.003769 01100101

49 0.004425 10110001

50 0.003754 01100100

51 0.004028 10000111

52 0.004440 10110100

53 0.003860 01110111

54 0.004333 10101010

55 0.004349 10101101

56 0.004044 10001011

57 0.004257 10100110

58 0.004150 10100010

59 0.004028 10001000

60 0.003586 01010001

61 0.003845 01110001

62 0.003983 10000010

63 0.003998 10000011

64 0.003784 01101101

65 0.004303 10100111

66 0.004456 10110111

67 0.004440 10110101

68 0.004364 10101110

69 0.004379 10101111

70 0.004929 11101010

71 0.005692 0001011

72 0.005600 0000110

73 0.006073 0010111

74 0.005966 0010100

75 0.007278 0101111

76 0.008102 1000110

77 0.009796 1110001

78 0.008255 1001101

79 0.008179 1001010

80 0.007935 1000000

81 0.007812 0111110

82 0.007233 0101100

83 0.006561 0100001

84 0.005630 0001000

85 0.005722 0001100

86 0.005630 0001001

87 0.004974 11101110

88 0.005081 11110000

89 0.004959 11101101

90 0.005417 11111110

91 0.005157 11110100

92 0.005280 11111011

93 0.005783 0001111

94 0.005737 0001101

95 0.005829 0010001

96 0.006073 0011000

97 0.005997 0010101

98 0.006149 0011001

99 0.007095 0101001

100 0.006516 0100000

101 0.007156 0101010

102 0.007706 0111010

103 0.008438 1010010

104 0.008911 1011100

105 0.009216 1101010

106 0.009003 1011110

107 0.009109 1100011

108 0.009048 1100001

109 0.008209 1001011

110 0.008026 1000010

111 0.006454 0011110

112 0.006714 0100101

113 0.006668 0100010

114 0.006805 0100110

115 0.007156 0101011

116 0.007248 0101101

117 0.008133 1001000

118 0.008316 1001110

119 0.008331 1001111

120 0.009232 1101011

121 0.009247 1101100

122 0.009506 1101111

123 0.009109 1100100

124 0.008636 1010100

125 0.007797 0111101

126 0.007675 0111001

127 0.007294 0110000

128 0.007538 0110100

129 0.007263 0101110

130 0.008331 1010000

131 0.009003 1011111

132 0.009140 1100101

133 0.009155 1100110

134 0.009796 1110010

135 0.008865 1011001

136 0.009003 1100000

137 0.009186 1101000

138 0.007904 0111111

139 0.007309 0110001

140 0.006683 0100011

141 0.006805 0100111

142 0.005554 0000011

143 0.004944 11101011

144 0.005295 11111100

145 0.004562 11001110

146 0.005417 11111111

147 0.004456 10111010

148 0.004211 10100011

149 0.004333 10101011

150 0.004028 10001001

151 0.003769 01101010

152 0.003769 01101011

153 0.004120 10011001

154 0.004456 10111011

155 0.004684 11011011

156 0.003860 01111000

157 0.004059 10010010

158 0.004440 10110110

159 0.004395 10110000

160 0.003494 01010000

161 0.003845 01110110

162 0.002930 00100101

163 0.002579 111100011

164 0.002808 00001111

165 0.002533 111100010

166 0.002396 111000001

167 0.002060 100110001

168 0.001999 100010100

169 0.002045 100110000

170 0.001968 011110011

171 0.002319 110011111

172 0.001907 011100000

173 0.002121 101011000

174 0.002213 101011001

175 0.002625 111110010

176 0.002747 00000100

177 0.002640 111110011

178 0.002975 00100111

179 0.003220 00111110

180 0.002792 00000101

181 0.003113 00111001

182 0.002930 00100110

183 0.002441 111000010

184 0.002472 111011000

185 0.002609 111101010

186 0.002853 00010101

187 0.002792 00001110

188 0.002640 111111010

189 0.002701 111111011

190 0.002899 00011101

191 0.002808 00010100

192 0.002731 00000010

193 0.002991 00101100

194 0.003296 01001000

195 0.003143 00111010

196 0.003174 00111011

197 0.003098 00110101

198 0.003052 00101101

199 0.002869 00011100

200 0.003098 00111000

201 0.002899 00100100

202 0.002609 111101011

203 0.001907 011100001

204 0.001907 011110010

205 0.001572 001101001

206 0.001144 1100111101

207 0.001160 1110000000

208 0.001190 1110110010

209 0.001160 1110000001

210 0.000488 10001010100

211 0.000763 0011010001

212 0.000687 0000001110

213 0.000687 0000001111

214 0.000458 01001001111

215 0.000519 10001010110

216 0.000244 100010101011

217 0.000320 111011001101

218 0.000275 100010101111

219 0.000214 001101000011

220 0.000259 100010101110

221 0.000076 11101100110011

222 0.000046 00110100000111

223 0.000137 1110110011000

224 0.000046 00110100001000

225 0.000046 00110100001001

226 0.000031 001101000010111

227 0.000015 001101000001000

229 0.000015 001101000001001

231 0.000031 111011001100100

240 0.000015 001101000010100

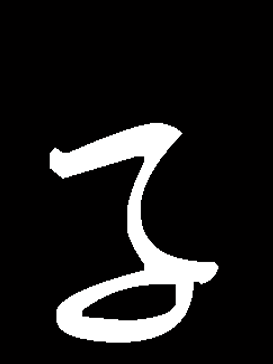
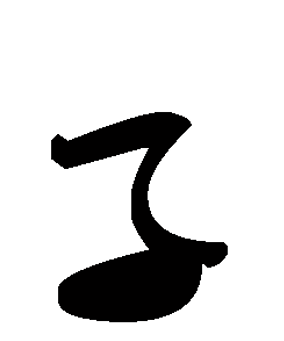
Average Length: 7.459305

Compression Ratio:1.072486

即编码平均长度约为7.46，压缩比为1.07。

## 题目二

针对图像regiongrowing.bmp，编写基于区域增长的图像分割算法，当种子点分别选取在前景和背景时，要求得到如下结果图，同时需要贴出两张区域增长过程中的中间效果图。

种子选在前景 种子选在背景

### 题目分析

区域增长算法是指从一组“种子”点开始，将与种子预先定义的性质相似的那些邻域像素添加到每个种子上来形成这些生长区域。我们可以通过队列，将每次处理的点附近与之差异不大的点放入处理队列中，由此从种子点进行区域扩散并分割图像。

### 代码实现

*% 读取图像*

I = imread('regiongrowing.bmp');

*% 显示原始图像*

figure;

subplot(2, 3, 1);

imshow(I);

title('Original Image');

*% 设置种子点*

seed1 = [125, 100]; *% 前景种子点*

seed2 = [200, 200]; *% 背景种子点*

*% 定义阈值*

threshold = 20;

subplot(2, 3, 4);

*% 基于第一个种子点的区域增长*

segmented1 = myRegionGrowing(I, seed1, threshold);

*% 显示第一个种子点的区域增长效果图*

subplot(2, 3, 2);

imshow(segmented1);

title('Segmentation with Seed 1');

subplot(2, 3, 5);

*% 基于第二个种子点的区域增长*

segmented2 = myRegionGrowing(I, seed2, threshold);

*% 显示第二个种子点的区域增长效果图*

subplot(2, 3, 3);

imshow(segmented2);

title('Segmentation with Seed 2');

**function** **segmented** = **myRegionGrowing**(I, seed, threshold)

*% 初始化输出图像*

segmented = zeros(size(I));

[rows, cols] = size(I);

*% 访问标记数组，标记已经添加到区域中的像素*

visited = false(size(I));

*% 队列用于存储待处理的像素坐标*

queue = zeros(rows\*cols, 2);

queueSize = 0;

*% 添加种子点到队列中*

queueSize = queueSize + 1;

queue(queueSize, :) = seed;

*% 区域增长过程*

**while** queueSize > 0

*% 从队列中取出当前像素的坐标*

currentPixel = queue(queueSize, :);

queueSize = queueSize - 1;

x = currentPixel(1);

y = currentPixel(2);

*% 检查当前像素是否已经访问过*

**if** visited(x, y)

**continue**;

**end**

*% 标记当前像素为已访问*

visited(x, y) = true;

*% 将当前像素添加到分割结果中*

segmented(x, y) = 1;

*% 检查当前像素的相邻像素*

**for** dx = -1:1

**for** dy = -1:1

*% 计算相邻像素的坐标*

nx = x + dx;

ny = y + dy;

*% 检查相邻像素是否在图像范围内*

**if** nx < 1 || nx > rows || ny < 1 || ny > cols

**continue**;

**end**

*% 检查相邻像素是否已经访问过*

**if** visited(nx, ny)

**continue**;

**end**

*% 计算当前像素与相邻像素的灰度差*

diff = abs(int32(I(nx, ny)) - int32(I(x, y)));

*% 如果灰度差小于阈值，则将相邻像素添加到队列中*

**if** diff <= threshold

queueSize = queueSize + 1;

queue(queueSize, :) = [nx, ny];

**end**

**end**

**end**

*% 每经过一定数量的迭代后，显示中间效果图*

**if** mod(queueSize, 100) == 0

imshow(segmented);

drawnow;

**end**

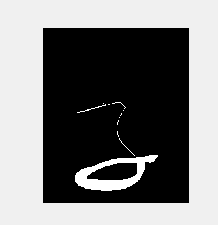
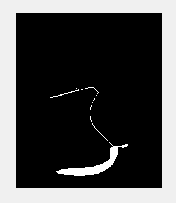
**end**

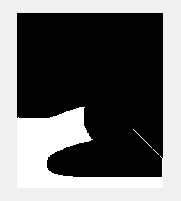
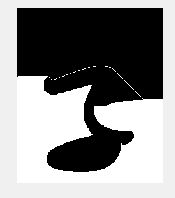
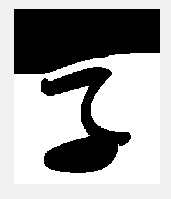
**end**

### 结果展示

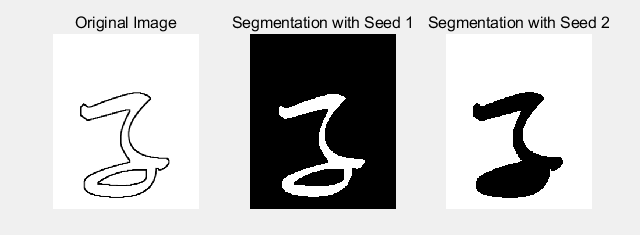
该程序会实时展示图像分割的进度（即中间效果图）。

中间效果图如下：



最终效果图：



## 总结

在本次作业中，我实现了使用霍夫曼编码进行图像压缩和使用区域增长的图像分割算法。图像压缩和图像分割都是图像处理中的重要内容，实际使用的图像压缩或图像分割算法也往往复杂得多。