Delft University of Technology

Data compression Huffman Coding
EE4740

Data compression Huffman Coding

Authors:
LongXIN (4799996)

March 17, 2024

1 Overall Code Design

1.1 Design ideas

The Huffman encoding process involves following steps

- mapping the audio signal to quantization levels
- calculating the probabilities of these levels
- filtering out levels with zero probability
- generates a Huffman tree based on the frequency of each quantization level
- using the resulting Huffman code to encode the quantized signal
- calculating the efficiency of the Huffman coding process

The Huffman decoding process decodes a given encoded audio string using the provided Huffman code map and involves only a few steps.

- iterating over the encoded string, matching each segment to a Huffman code in the code map
- converting it back to the corresponding quantization level
- the process continues until the entire encoded string has been decoded, or until a segment with no matching Huffman code is encountered, which triggers a warning.

1.2 Code Implementation

Since matlab's mlx format is used, relevant comments can be organically combined in the code, so the code is highly readable. Please move to the code part to view the specific implementation. The required code is implemented in detail at 2.2

2 Number of Huffman code entries

2.1 Theoretical Analysis

The trade-off between the number of Huffman code entries and computational complexity (including encoding and decoding speed) is a key aspect of this project. Overall, as the number of Huffman code entries increases, we can recover more refined sounds, but the computational complexity also increases.

Huffman Table Size
The size of Huffman table is proportional to the number of number of Huffman code entries or the quantized level L. More Huffman code entries means a larger Huffman table. It will result in better fidelity to the original data.

Computational Complexity
Space complexity
The size of the Huffman table directly determines the amount of information that must be stored and processed during encoding and decoding. Larger tables require more memory space.
Time complexity
Encoding time: During the encoding process, the Huffman table must be searched to find the encoding corresponding to the original data symbols. The larger the table, the longer the average lookup time is likely to be.

Decoding time: During decoding, recovering the original data from the encoded data usually involves traversing the Huffman tree. The size and depth of the tree are proportional to the size of the Huffman table, thus affecting decoding speed.

2.2 Piratical Measurement

Criteria
The test is based on audio set SA1.wav, the test is based on different quantization level L. The relation of Huffman code entries and consumed time is shown as below.

$$efficient = \frac{average\ length}{entropy} \tag{1}$$

Code overview
For L from 5 to 200, take the average running time of 3 running each time to measure the complexity of the algorithm.

Result Plot

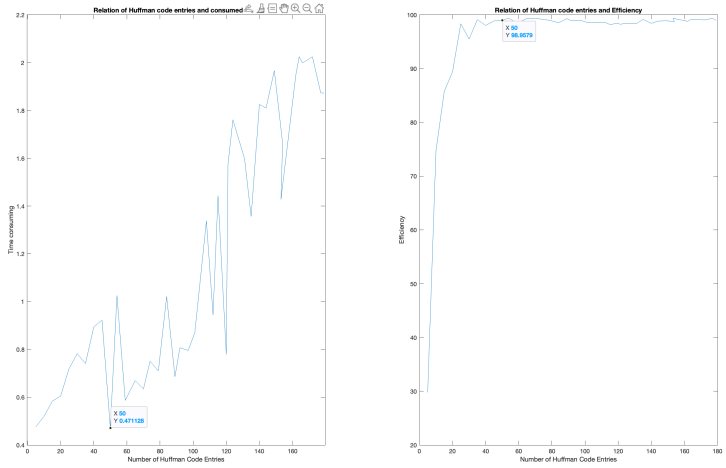

Figure 1: Relation of Huffman code entries and consumed time and Efficiency

From the Figure 1, we can see that Starting from L 25, the efficiency has reached 98%, and the running time begins to increase amid fluctuations. Based on what is happening, when L is 50, the efficiency reaches 98.95, with a high efficiency and low running time. So we should chosse L as 50.

However, as the Number of Huffman Code Entries increases, the decrease in computer runtime is unusual. Some possible reasons for this include:

- Characteristics of the test data: If the data set used for testing has different characteristics (such as different character distributions) at different test points, then even if the number of code table entries increases, the actual time required for encoding or decoding may be due to the characteristics of the data fluctuate due to differences.
- Caching and other system effects: In an actual computing environment, factors such as CPU cache hit rate and memory access speed may also affect the execution time of the algorithm. Especially when processing large amounts of data, these system-level effects can cause fluctuations in time consumption.
- Error in time measurement: The measurement of execution time may be interfered by a variety of factors, including system load, multi-task competition for resources, etc., resulting in fluctuations in the execution efficiency of the same operation at different times.

3 Training data

In the case of speech signals, training data refers to the collection of speech samples used to construct the Huffman table. In theory, these data should be representative, i.e. they should cover the possible range of variations in human speech, so that the constructed Huffman table can encode the actual speech signal as efficiently as possible. "Representative" here can be achieved by including speech data from different people

(gender, age, accent, etc.) and different situations (emotion, speed, background noise, etc.).
The data set we have is 8 audio messages of men and 8 women.

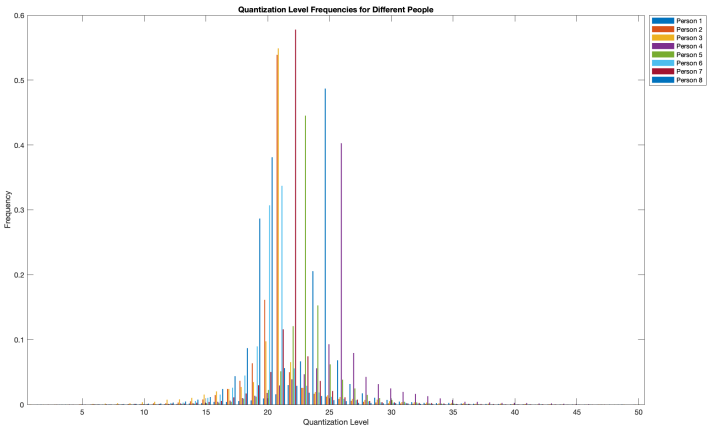

Figure 2: Quantization Level Frequencies for Different People

Figure 2 shows the frequency of speech of different people at various levels of quantification. As can be seen from the figure, different individuals have significantly different distribution patterns at different levels of quantification. These differences may be related to factors such as the pitch, timbre or dynamic range of the speech.
Our general idea is to use as many different people as possible, so we at least use every voice of each person, which covers the gender differences between men and women and the possible differences between different people. What's more, according to machine learning experience, we can use up to 80% of the data for training and 20% of the data for testing.
Based on the above reasons, we can select one male voice and one female voice as test data, and the others (14 in total) as training data.
In the code, I wrote the code based on 14 sets of training audios, and used 2 sets of test audios to encode and restore based on the Huffman tree of the training audios. The results show complete recovery within the computational error.

4 Robustness

One of the main features of Huffman coding is the efficiency of the encoding, but this also brings a disadvantage: sensitivity to bit loss. In the Huffman encoded data stream, the encrypted data is a long code composed of 0s and 1s. If a bit is lost, all data from the loss point to the next clear synchronization point will be lost. May not be decoded correctly. This is because Huffman coding relies on a complete data sequence for decoding, and missing bits will cause errors in the path during the decoding process, thus affecting the interpretation of all subsequent codes.

Some possible solutions are as follows,

- Add synchronization signals: regularly insert synchronization sequences or special synchronization words that can be easily recognized in the encoded data. In this way, even if data loss occurs, the decoder can resynchronize when encountering the next synchronization signal, thus only affecting the data between the lost bit and the next synchronization point.

- Error detection and correction: Introduce error detection and correction mechanisms such as cyclic redundancy check or Hamming code. These mechanisms can help detect errors in the data and correct them in some cases.

- Data segmentation and retransmission: Divide the data into smaller segments and append certain verification information to each segment. The receiver can check the integrity of each segment and request

retransmission of corrupted segments.

Here we have no control over the signal sending and receiving modes, so I chose to add the Hamming code recognition mode to the encoded code string.
This function is accomplished with the help of MALTAB encode and decode functions. It must be noted here that due to the memory limitations of my personal computer, I cannot encode the entire real data, so here I only use a part of the string obtained by Huffman encoding of the real audio to complete the Hamming code perturbation test.

For applying Hamming code to the result string of Huffman encoding, n (codeword length) has a certain relationship with k (message length) as Equation 2. This requires us to first add 0 to our information to complete the Hamming code encoding.

$$\begin{aligned} n &= 2^m - 1 \\ k &= n - m \end{aligned} \tag{2}$$

After reasonably expanding our string, we can use the *encode* and *decode* functions to encode the string with Hamming code. In fact, this greatly expands the matrix, which is why my personal computer does not have enough memory.

Since this is just a demonstration of the importance of Hamming codes for robustness, it is not embedded in the code. It should be noted that the encryption of Hamming code should be completed after the encryption of Huffman code, and the decryption of Hamming code should be before decoding of Huffman code.

Experiments show that this process effectively prevents interference problems that may occur during transmission. Even if some transmission contents are incorrect, we can still recover the complete and correct Huffman code based on the Hamming code.

5 Code

Related code and files can be found on my GitHub too
https://github.com/EPIGP/Huffman-Coding-EE4740

```
clc
clear

% addpath("EE4740_Miniproject_1/")
sound = audioread("SA1.wav");
```

Question 1

Preliminary research, this part aims to study the selection of L suitable for Huffman code

```
num = 40;
t = zeros(1,num);

h=waitbar(0,'please wait');

for i = 1:num
    for j = 1:3
        L = i*5;
        tStart = tic;
        [quantized_sound, level, encoded_audio_string, codeMap, efficiency]
= HuffmanCoding(sound, L);
        decoded_audio = HuffmanDecoding(encoded_audio_string, codeMap);

        tolerance = 1e-5;
        Check(quantized_sound, decoded_audio, tolerance);
        time_buffer(j) = toc(tStart);
        eff_buffer(j) = efficiency;
        waitbar(i/num, h)
    end
    t(i) = mean(time_buffer);
    eff(i) = mean(eff_buffer);
    entries(i) = size(codeMap,1);
end
delete(h);

subplot(1,2,1)
plot(entries, t)
xlabel('Number of Huffman Code Entries')
ylabel('Time consuming')
title('Relation of Huffman code entries and consumed time')

subplot(1,2,2)
plot(entries, eff)
xlabel('Number of Huffman Code Entries')
ylabel('Efficiency')
title('Relation of Huffman code entries and Efficiency')
```

Question 2

2.1

This section aims to show the differences between different audio data

```
L = 50;

audio_files = {'person1.wav', 'person2.wav', 'person3.wav', 'person4.wav',
'person5.wav', 'person6.wav', 'person7.wav', 'person8.wav'};

quant_freqs = zeros(L, length(audio_files));

for i = 1:length(audio_files)
    [sound, ~] = audioread(audio_files{i});

    max_ = max(sound);
    min_ = min(sound);
    step = (max_ - min_) / (L-1);
    level = min_ : step : max_;
    differences = abs(sound - reshape(level, [1, 1, length(level)]));
    [~, indices] = min(differences, [], 3);
    quantized_sound = level(indices)';

    for j = 1:L
        quant_freqs(j, i) = sum(quantized_sound == level(j)) /
length(quantized_sound);
    end
end

figure
bar(quant_freqs, 'grouped')
title('Quantization Level Frequencies for Different People')
xlabel('Quantization Level')
ylabel('Frequency')
legend('Person 1', 'Person 2', 'Person 3', 'Person 4', 'Person 5', 'Person
6', 'Person 7', 'Person 8', 'Location', 'bestoutside')
```

2.2

This part is the required Huffman algorithm implementation. Note that robustness will be considered separately in Question 3, reason shown the same place.

```
clc
clear

training_files = {'person2.wav', 'person3.wav', 'person4.wav', ...
                  'person6.wav', 'person7.wav', 'person8.wav', ...
                  'person12.wav', 'person22.wav', 'person32.wav', ...
                  'person42.wav', 'person52.wav', 'person62.wav', ...
                  'person72.wav', 'person82.wav'};
```

```
test_files = {'person1.wav', 'person5.wav'};

training_data = [];
for i = 1:length(training_files)
    [data, Fs] = audioread(training_files{i});
    if i == 1
        training_Fs = Fs;
    else
        if Fs ~= training_Fs
            error('Sampling rate mismatch in training data.');
        end
    end
    training_data = [training_data; data];
end

test_data = [];
for i = 1:length(test_files)
    [data, Fs] = audioread(test_files{i});
    if i == 1
        test_Fs = Fs;
    else
        if Fs ~= test_Fs
            error('Sampling rate mismatch in test data.');
        end
    end
    test_data = [test_data; data];
end

L = 50;
[~, level, encoded_audio_string, codeMap, Efficiency] =
HuffmanCoding(training_data, L);
quantized_test_data = zeros(size(test_data));

%Test
test_data(test_data == 0) = []; % Remove the silent part, if any
quantized_test_data = quantizeSound(test_data, level);

encoded_test_string = HuffmanEncode(quantized_test_data, codeMap);
decoded_test_data = HuffmanDecoding(encoded_test_string, codeMap);

tolerance = 1e-5;
% Check if the two arrays are exactly equal
are_equal = isequal(quantized_test_data, decoded_test_data);

errors = abs(quantized_test_data - decoded_test_data);
are_close = all(errors < tolerance);

disp(['Arrays are equal: ', num2str(are_equal)]);
```

```
Arrays are equal: 0
```

```
disp(['Arrays are close within tolerance: ', num2str(are_close)]);
```

```
Arrays are close within tolerance: 1
```

Question 3

Due to personal computer memory limitations, it is impossible to encrypt whole audio data with Hamming code. Here I only use the part of the Huffman encoding string of audio data as a demonstration.

```
clc
clear
sound = audioread("SA1.wav");

L = 50;
[quantized_sound, level, encoded_audio_string, codeMap, Efficiency] =
HuffmanCoding(sound, L);

encoded_audio_string = encoded_audio_string(1:10000);
[padded_vector, num_added_zeros, n, k] =
padEncodedString(encoded_audio_string);

encData = encode(padded_vector,n,k,'hamming/binary');

%Add perturbation
errLoc = randerr(1,n);
encData = mod(encData + errLoc',2);

decData = decode(encData,n,k,'hamming/binary');

str_without_padding = removePadding(decData, num_added_zeros);

if length(str_without_padding) == length(encoded_audio_string) &&
all(str_without_padding == encoded_audio_string)
    disp('The string perturbed by the Hamming code has not changed.')
end
```

```
The string perturbed by the Hamming code has not changed.
```

Function definition

```
function [padded_vector, num_added_zeros, n, k] =
padEncodedString(encoded_test_string)
    % Check and convert the input string
    if isnumeric(encoded_test_string) || islogical(encoded_test_string)
        % Assume encoded_test_string is a binary array, convert it to a
character array
        encoded_test_string = num2str(encoded_test_string(:).');
        encoded_test_string = strrep(encoded_test_string, ' ', ''); %
Remove the spaces added by num2str
```

```
    end

    original_length = length(encoded_test_string);

    m = 2;
    while (2^m - m - 1) < original_length
        m = m + 1;
    end

    n = 2^m - 1;
    k = n - m;

    num_complete_info_bits = ceil(original_length / k) * k;

    num_added_zeros = num_complete_info_bits - original_length;

    padded_string = [encoded_test_string, repmat('0', 1, num_added_zeros)];

    %Adjust data format
    padded_vector = padded_string - '0';
    padded_vector = padded_vector';
end

function [data_str] = removePadding(decData, num_added_zeros)
    if num_added_zeros > 0
        data_without_padding = decData(1:end-num_added_zeros);
    else
        data_without_padding = decData;
    end

    data_str = num2str(data_without_padding');
    data_str = data_str(~isspace(data_str));
end

function [quantized_sound2, level, encoded_audio_string, codeMap,
Efficiency] = HuffmanCoding(sound, L)
    max_ = max(sound);
    min_ = min(sound);
    step = (max_ - min_) / (L-1);

    level = min_ : step : max_;

    % Map the sound value to the corresponding level
    differences = abs(sound - reshape(level, [1, 1, length(level)]));
    [~, indices] = min(differences, [], 3);
    quantized_sound2 = level(indices)';

    % Get the probabilities of different levels
    prob2 = zeros(size(level));
    for i = 1:length(level)
```

```
        prob2(i) = sum(quantized_sound2 == level(i));
    end
    prob2 = prob2 / length(quantized_sound2);

    % Remove those value with 0 probability, and resort value.
    sorted_matrix = sortrows([prob2; level]', 1, 'descend')';
    sorted_prob2 = sorted_matrix(1, :);
    sorted_level = sorted_matrix(2, :);

    nonzero_prob = sorted_prob2(sorted_prob2 > 0);
    nonzero_level = sorted_level(1:size(nonzero_prob,2));

    num = nnz(nonzero_prob);

    % Applying Huffman coding
    % prob = [0.25, 0.25, 0.2, 0.15, 0.15];  % Example Probability
    % symbols = {'A', 'B', 'C', 'D', 'E'};  % Example Symbols
    prob = nonzero_prob;
    symbols = arrayfun(@(x) num2str(x), nonzero_level, 'UniformOutput',
false);

    num = nnz(prob);
    HuffmanTree = cell(num, 1);

    for i = 1:num
        HuffmanTree{i} = {symbols{i}, prob(i), {}, {}, {}};  % Each node :
{Symbol, Prob, LeftNode, RightNode, Huffman Code}
    end

    % Creating Huffman Tree
    while length(HuffmanTree) > 1
        [~, order] = sort(cellfun(@(x) x{2}, HuffmanTree), 'descend');
        HuffmanTree = HuffmanTree(order);

        newNode = {[], HuffmanTree{end}{2} + HuffmanTree{end-1}{2},
HuffmanTree{end}, HuffmanTree{end-1}, {}};

        % Add new node and remove the last node
        HuffmanTree{end-1} = newNode;
        HuffmanTree(end) = [];
    end

    % Generate Huffman code
    HuffmanTree = HuffmanCode(HuffmanTree);  % From the root node

    % Calculate Entropy
    H = 0;
    for i = 1:num
        H = H - prob(i)*log2(prob(i));
    end
```

```
    % Encode the speech signal
    symbolCodes = ExtractHuffmanCodes(HuffmanTree);
    Length = 0;
    for i = 1:num
        prob = symbolCodes{i, 2};  % Extract probability
        codeLength = length(symbolCodes{i, 3});  % Calculate the length of
the Huffman code
        Length = Length + (prob * codeLength);  % Accumulate probability
multiplied by code length
    end

    % Calculate the maximum efficiency of Huffman code for this speech
signal
    Efficiency = (H / Length)*100;

    % Encode the speech signal
    encoded_audio = cell(length(quantized_sound2), 1);
    codeMap = cell(num, 2);

    for i = 1:num
        codeMap{i, 1} = symbolCodes{i, 1};  % Quantization levels
        codeArray = cell2mat(symbolCodes{i, 3});  % Convert 1x4 cell to 1x4
array
        codeStr = num2str(codeArray);  % Convert numeric array to string
        codeStr = strrep(codeStr, ' ', '');  % Remove spaces
        codeMap{i, 2} = codeStr;  % Huffman encoding
    end

    for i = 1:length(quantized_sound2)
        levelStr = num2str(quantized_sound2(i));  % Convert quantization
level to string
        for j = 1:num
            if strcmp(codeMap{j, 1}, levelStr)
                encoded_audio{i} = codeMap{j, 2};  % Add corresponding
Huffman code to the result cell array
                break;
            end
        end
    end

    % Iterate through each element in encoded_audio
    encoded_audio_string = strcat(encoded_audio{:});  % Concatenate into
the final string

    % fprintf('Length of compression is: %d\n',
length(encoded_audio_string));
end
```

```
% Decode the speech signal
function decoded_audio = HuffmanDecoding(encoded_audio_string, codeMap)
    decoded_audio = [];
    start_index = 1;

    while start_index <= length(encoded_audio_string)
        match_found = false; % Flag
        for i = 1:size(codeMap,1)
            huffmanCode = codeMap{i,2};
            codeLength = length(huffmanCode);
            if start_index + codeLength - 1 <= length(encoded_audio_string)
&& ...
                    strcmp(encoded_audio_string(start_index:start_index +
codeLength - 1), huffmanCode)

                decoded_audio = vertcat(decoded_audio,
str2double(codeMap{i, 1}));
                start_index = start_index + codeLength;
                match_found = true;
                break; % Break the loop after a match
            end
        end

        if ~match_found
            warning('No match found for the sequence starting at index
%d.', start_index);
            break;
        end
    end
end


function Check(quantized_sound2, decoded_audio, tolerance)
    % Check if the two arrays are exactly equal
    are_equal = isequal(quantized_sound2, decoded_audio);

    % Or compare using a certain tolerance
    % tolerance = 1e-5;
    errors = abs(quantized_sound2 - decoded_audio);
    are_close = all(errors < tolerance);

    % disp(['Arrays are equal: ', num2str(are_equal)]);
    % disp(['Arrays are close within tolerance: ', num2str(are_close)]);
    if are_close == 0
        disp(['Arrays are not close within tolerance'])
    end
end


function nodes = HuffmanCode(nodes)  % Build Code Huffman Tree
```

```
    if length(nodes) == 1
        nodes = nodes{1}; % Extract rootnode
    end

    if ~isempty(nodes{3})  % There is left node
        nodes{3}{5} = [nodes{5} 0];
        nodes{3} = HuffmanCode(nodes{3});
    end

    if ~isempty(nodes{4})  % There is right node
        nodes{4}{5} = [nodes{5} 1];
        nodes{4} = HuffmanCode(nodes{4});
    end
end

function symbolCodes = ExtractHuffmanCodes(Nodes)
    if length(Nodes) == 1
        Nodes = Nodes{1};  % Extract rootnode
    end

    symbolCodes = {};%{Symbol, Prob, Huffman Code}

    % if this is a leaf node
    if isempty(Nodes{3}) && isempty(Nodes{4})
        symbolCodes = {Nodes{1}, Nodes{2}, Nodes{5}};
    else
        % Otherwise, recursively traverse the tree
        if ~isempty(Nodes{3})  % If there is a left child node
            codesLeft = ExtractHuffmanCodes(Nodes{3});  % Recursively
obtain the encoding of the left child node
            symbolCodes = [symbolCodes; codesLeft];  % Merge cell arrays
        end
        if ~isempty(Nodes{4})  % If there is a right child node
            codesRight = ExtractHuffmanCodes(Nodes{4});  % Recursively
obtain the encoding of the right child node
            symbolCodes = [symbolCodes; codesRight];  % Merge cell arrays
        end
    end
end

function quantized_sound = quantizeSound(sound, level)
    % Map the sound value to the corresponding level
    differences = abs(sound - reshape(level, [1, 1, length(level)]));
    [~, indices] = min(differences, [], 3);
    quantized_sound = level(indices)';
end

function encoded_audio_string = HuffmanEncode(quantized_sound, codeMap)
    encoded_audio = cell(length(quantized_sound), 1);
```

```
    for i = 1:length(quantized_sound)
        levelStr = num2str(quantized_sound(i));

        for j = 1:size(codeMap, 1)
            if strcmp(codeMap{j, 1}, levelStr)
                encoded_audio{i} = codeMap{j, 2};
                break;
            end
        end
    end

    encoded_audio_string = strcat(encoded_audio{:});
end
```