Masaryk University
Faculty of Informatics

Pure Java WSQ Encoder

Master's Thesis

Sebastián Lazoň

Brno, Spring 2017

Masaryk University
Faculty of Informatics

Pure Java WSQ Encoder

Master's Thesis

Sebastián Lazoň

Brno, Spring 2017

This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Sebastián Lazoň

Advisor: Ing. Petr Adámek

Acknowledgement

I would like to thank the advisor Ing. Petr Adámek for his time, for bringing up the topic and providing helpful input as well as my colleagues for assistance during the testing and integration process. I would also like to thank my family for support and encouragement.

Abstract

This thesis focuses on fingerprint image compression using the Wavelet Scalar Quantization (WSQ) algorithm. The primary goal is to implement the WSQ image encoder in Java programming language, test it against the specification and integrate it with the existing fingerprint processing solution of Home Credit Group.

The textual part contains overview of commonly used image compression methods, detailed definition of the WSQ encoding process and the practical part's implementation and testing process description including the evaluation.

Keywords

wavelet scalar quantization, discrete wavelet transform, Huffman coding, image compression, fingerprints, biometrics, Java

Contents

List of Tables

List of Figures

1 Introduction

Fingerprint recognition is one of the most widely used methods of biometric authentication, offering good compromise between the price of sensors and provided accuracy. These characteristics combined with simple extraction process make it popular choice for mass applications. However, process of extracting multiple fingerprints for each person generates significant amount of data which then has to be transferred, processed and stored. Some form of compression is therefore usually used to reduce the storage space and data transfer times. Lossy image compression algorithm named Wavelet Scalar Quantization (WSQ) was developed specifically for this purpose by the Federal Bureau of Investigation (FBI) and has became a standard for storage and exchange of fingerprint images.

Client identification solution of Home Credit group is also using it internally. A Java application is used for fingerprint capturing and WSQ support is provided by biometric library written in C. However, this library is currently causing problems with stability, portability and footprint of the application (see chapter 4 for details). This led to demand for an alternative written in Java which was satisfied by the practical part of this thesis.

The goals

The primary objective of this thesis is to create pure Java WSQ encoder based on NIST reference implementation [1]. Output must meet the WSQ specification and performance testing should be executed to verify its system demands. Textual part should contain description of the idea behind the WSQ, explanation of the encoding process and the resulting file structure. Description of the implementation and testing process including the results should also be included.

Overview

The second chapter briefly describes the theory of data compression and its specifics when used for image data. It also provides overview of the most used compression methods and related algorithms used in practice with their advantages and disadvantages.

The third chapter contains detailed description of methods mentioned before, used in WSQ algorithm. Definition of internal data structures, explanation of the encoding process and output file structure according to specification are also included.

The fourth chapter contains description of Home Credit Fingerprint Solution's architecture and its integration with the WSQ encoder. More details regarding the motivation are also provided.

The fifth chapter talks about the process of Java WSQ encoder's implementation and differences between the given programming languages which had to be faced throughout.

The sixth chapter describes the process of Java WSQ encoder's testing, verification of both that the implementation is returning the correct data and that computational complexity while doing so is acceptable for intended application. At the end, evaluation of tests is provided.

The last chapter contains conclusion, evaluation of the objectives and list of possibilities of future extension.

2 Image compression

Image representation and processing is area where data compression is of great importance. Especially with technology advancements in visual data capturing and growing Internet usage, means of efficient storage and transmission of visual data have been subject of intensive research. Data compression is the process of reducing amount of data required to represent given quantity of information by eliminating redundancy. The most intuitive way of encoding visual data is two-dimensional array of intensity values. This representation generally suffers from these types of data redundancy [2]:

Coding Redundancy

Coding redundancy is related to the probability distribution of individual values in the data. This distribution is rarely uniform as in most images some pixel values are more common than the others. Assume an grayscale image where each pixel is assigned an value <0-255>. According to [2], the average number of bits required to represent a pixel is:

$$p_r(r_k) = \frac{n_k}{MN} \tag{2.1}$$

$$L_{avg} = \sum_{k=0}^{L-1} l(r_k) p_r(r_k) \tag{2.2}$$

Where M, N are dimensions of the image, L is a number of intensity values, r_k discrete random variable representing intensities of the image, $l(r_k)$ number of bits used to represent given intensity and n_k number of times the given intensity appears in the image. Natural binary coding assigns fixed-length code (8-bit integer in this case) to each pixel value regardless of the number of occurrences in the given image. Variable length code schemes try to minimize this number by exploiting nonuniform probabilities of value occurrences and by assigning shorter codewords to the more common values. Examples include Huffman coding (see section 2.1) and Arithmetic coding (see section 2.1).

Interpixel Redundancy

In most images, certain degree of correlation (color, intensity) exists between some neighbor pixels (both x and y dimension), therefore amount of useful information holden by them is not the same across the image. The value of the pixel can be inferred from values of its surroundings to some level. To reduce this redundancy, 2D array has to be transformed to more efficient representation. This transformations are called mappings and can be either reversible or irreversible. Examples are Run-length encoding (see section 2.1) and LZW coding (see section 2.1).

Psychovisual Redundancy

Researches show that human eye sensitivity varies across color spectrum and intensity [3]. Therefore useful information for human eye holden by a pixel varies depending on its color and brightness. The goal is to remove information ignored by human visual system. However, this results in loss of quantitative information (referred as Quantization) so it is always irreversible operation. Examples are DCT (see section 2.1) and DWT (see section 2.1).

The Theory

Information theory based on work of Claude Shannon [4] provides theoretical background for the concept of data compression. It models communication systems using five elements:

Information Source

The producer of messages in given format for receiver.

Transmitter

Operates on message in order to produce signal suitable for transmission over the channel.

Channel

Medium used to transmit the signal from transmitter to receiver.

Receiver
Operates on received signal to reconstruct the original message.

Destination
Intended recipient of the message.

Entropy is used as the key measure denoting the average expected value of information contained in the message. When the definition is applied to the general image defined in formula 2.1, outcome is:

$$H = -\sum_{k=0}^{L-1} p_r(r_k) log(p_r(r_k)) \tag{2.3}$$

where the variable definitions are the same as in formula 2.1. As H is the average value of information per intensity in bits, it is the lower-bound of achievable compression – it is not possible to encode intensity values of the given image with fewer bits per pixel than this value.

Measuring information

From the previously mentioned, it is clear that in order to reduce the size even further, some part of the useful information has to be omitted. Assume $f(x,y)$ is input image, $\hat{f}(x,y)$ input image with part of information removed and $e(x,y) = \hat{f}(x,y) - f(x,y)$ the error. For accurate representation, following measures are used: [2]

Mean square error (MSE)

$$e_{rms} = \sqrt{\frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} [e(x,y)]^2} \tag{2.4}$$

Mean square signal-to-noise-ratio (SNR)

$$SNR_{ms} = \frac{\sum_{x=0}^{M-1} \sum_{y=0}^{N-1} [\hat{f}(x,y)]^2}{\sum_{x=0}^{M-1} \sum_{y=0}^{N-1} [e(x,y)]^2} \tag{2.5}$$

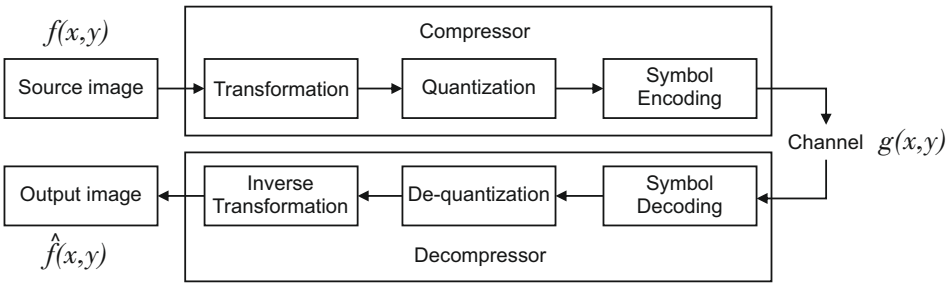

Figure 2.1: Image compression model

Image compression model

Most image compression systems follow schema described in figure 2.1, defining two components: Encoder, responsible for compressing the source and Decoder, executing the inverse operation. Input image $f(x, y)$ is processed by encoder and creates its compressed representation $g(x, y)$ which can be used as input for decoder in order to produce the original image $\hat{f}(x, y)$ (or in case of lossy compression, its approximation with error $e(x, y)$) . Main components' description of the encoding process follows [2]:

Transformation
: This process addresses interpixel redundancy and transforms input data representation accordingly. It may reduce the data volume itself or just prepare it for the next steps. In case of video compression, it also uses previous/next frames to reduce temporal[1] redundancy.

Quantization (lossy compression only)
: Process of mapping from set of input values M to N where $|M| > |N|$ according to predefined criteria. This operation reduces psychovisual redundancy, is irreversible and is the core of basically all lossy compression algorithms. Two basic types of quantization exist: scalar quantization, performing the mapping on each of the input values separately and vector quantization, replacing block of input values

1. Subtype of interpixel redundancy where image data are correlated in time

with the vector from the codebook. The difference between input value and its quantizied couterpart is reffered to as quantization error.

Symbol encoding
Lossless data compression scheme which reduces coding redundancy by mapping the input to either fixed or variable-length code.

2.1 Overview of techniques

This section contains description of compression techniques used in the most common image compression standards.

Huffman Coding

Published by David A. Huffman in 1952 [5], it described process of finding a Huffman code for given data input. Huffman code is a type of binary prefix code[2] with minimum expected codeword length in terms of Shannon's first theorem (see formula 2.3). The algorithm for construction uses binary tree where each of the nodes contain weight[3] information, leaf nodes contain also symbol itself.

The process of finding a Huffman code is split in two parts:

Building the tree
First, all possible input symbols are used as the leaf nodes. Then, for two nodes with the smallest weight, new internal node as their parent is created, where the weight is the sum of weights of its two children. This is repeated ignoring all the nodes which already have a parent until only one node remains. This node is referred to as the root of Huffman tree.

Assigning the code
The next step starts in the root and assigns '0' to the left

2. bit string representing one symbol is never a prefix of the bit string representing any other symbol
3. frequency of appearance of given value

and '1' to the right child. This is repeated for each node of the tree. The Huffman code is then defined for all symbols as the path from the root to leaf node containing given symbol.

The decoding process then translates the stream of prefix codes to individual values by traversing the Huffman tree, which can either be part of the encoded data or pre-computed in decoder[4].

Even though Huffman expects frequencies of symbol occurrences as part of the input, probabilities of input symbols are rarely known in advance. Therefore, either the algorithm will have to process the input twice – once to determine the probabilities and second time for actual construction of Huffman tree – which is too slow for practical use, or adaptive Huffman coding is used. This method developed by Vitter and Knuth [6, 7] starts with the empty tree, which is then updated as the symbols are read and their probabilities (and therefore codes) are changing on the fly. However, this is not an issue since the encoder and decoder are synchronized in a way – they perform the same updates to the tree, therefore at given time, both of them use the same codes for given input symbol.

Advantages of Huffman coding algorithm are simple implementation, high speed and lack of patent coverage. The main disadvantage of the algorithm is that it finds optimal coding only when each of the symbols are encoded separately, while other methods can achieve better compression by encoding the sequences of input symbols instead [8].

Arithmetic Coding

Introduced in 1979 [8], arithmetic coding can be viewed as generalization of Huffman coding. Specifically, it will return the same result when probability of each of the symbol is the power of $\frac{1}{2}$. In the other cases, the optimal entropy bound (see formula 2.3) can be approached more closely since each of the symbols does not have to be assigned separate code with integral length. The message is encoded as a whole instead.

4. at the expense of compression efficiency

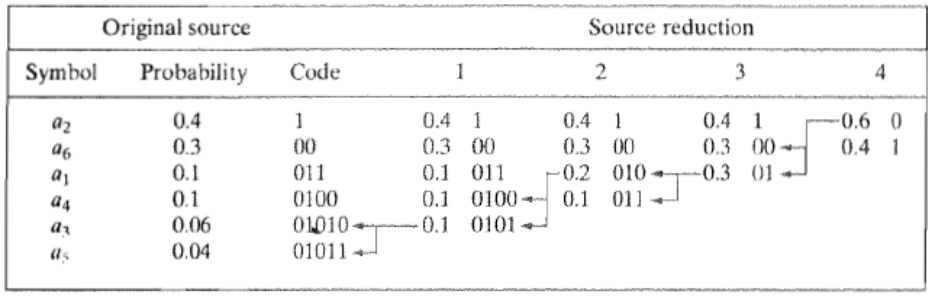

Figure 2.2: Huffman coding [2]

At first, statistical model of the input data has to be created. Its accuracy has significant effect on achieved compression rate. Adaptive model updates its prediction of symbol probabilities while reading the input data. Encoder starts by assigning the interval <0,1> to the input message. In every step, it holds current symbol of the input message, the current interval and probabilities assigned by the model. Once again, the encoding process can be split in two parts:

Finding the interval
: At the start, the interval <0,1> is assigned. For each input symbol, the interval is divided according to probability model and subinterval which corresponds to given symbol is used for the next iteration. When finished, the input message is represented by interval <a,b>.

Finding the representation
: Again, interval <0,1> is used at start, but now it is divided in half in every iteration. For every next iteration, subinterval <x,y> containing the <a,b> is chosen. This is repeated until the whole subinterval is contained in <a,b>. The binary representation of final interval's lower bound (x) is then used as the codeword for given input sequence.

The decoding process takes binary representation of the codeword (x) as input and divides the <0,1> interval according to probability model again. This is recursively repeated for subinterval containing the x. However, this process can be repeated indefinitely so a method

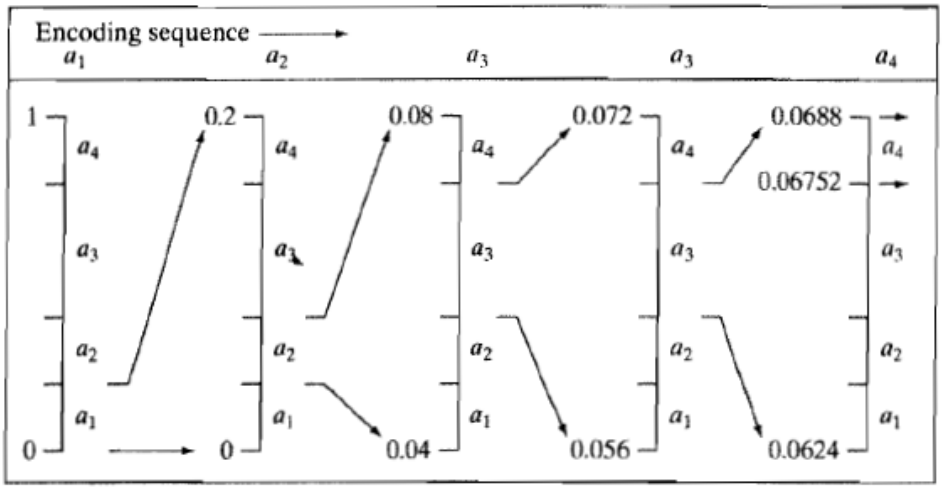

Figure 2.3: Arithmetic coding [2]

indicating when to stop must be used. Either length of the symbol sequence is part of the codeword or one of the symbols is designated as end-of-computation marker and when it is encountered, recursion stops. The original symbol sequence is then reconstructed from the path in the subintervals as shown in figure 2.3.

In 2013, faster alternative to Arithmetic coding was introduced with Asymmetric Numeral Systems [9]. This method combines the advantages of both Huffman and Arithmetic coding to produce high compression ratio with low computation cost. As this method is not employed by any commonly used image compression algorithm, it is not covered by this thesis.

LZW Coding

Lempel–Ziv–Welch (LZW) is a universal lossless data compression algorithm created by Abraham Lempel, Jacob Ziv, and Terry Welch in 1984 [10]. This technique assigns fixed-length codes to variable-length sequences of input symbols [2]. Important property of LZW coding is, that in contrast to Huffman coding, no a priori knowledge of input symbols' occurrence probabilities is needed.

In case of 8-bit input symbols, algorithm starts by creating dictionary of 256 items. Then, input symbols are sequentially examined. Every time new substring is discovered, it is added to the dictionary. When substring already contained in dictionary is discovered, the new input symbol is read and added to current string to create a new substring. This is then added to the dictionary. Each time string of symbols already contained in dictionary is encountered, it is encoded by its assigned code. The size of dictionary has to be limited in order not to grow indefinitely. However, if the size of the dictionary is too low, achievable compression ratio is reduced.

The decoding process reads the codeword and outputs its assigned symbol sequence. Every other consecutive codeword is then used to rebuild the dictionary the same way it was built by encoder.

Run-Length Coding

Run-length encoding (RLE) is a very simple form of lossless data compression which encodes runs of identical symbols as run-length pairs consisting of the symbol and length of the sequence [2]. It is evident that this method is effective when input symbols are repeated in long sequences, therefore it is suitable for bitmap images containing large areas of homogeneous color and intensity. However, it is also used in continuous-tone applications – JPEG uses RLE to encode coefficients of DCT after the quantization (see section 2.2).

Bit-plane Coding

The technique of bit-plane coding is based on decomposing of input image to series of binary images [2]. This decomposition is achieved when only certain bit position is taken into account of all the binary numbers representing pixel values. Separate coding of image's bit-planes reduces its inter-pixel redundancy.

In figure 2.4, it is evident that the first bit plane resembles the median value and the higher number of the bit-plane, the lower contribution to the final image. The first bit-planes also contain large homogeneous areas and are therefore suitable candidates for RLE.

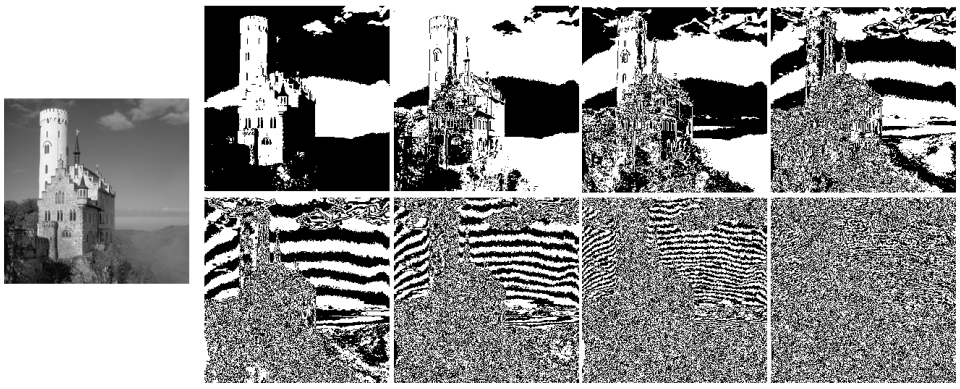

Figure 2.4: Bit-plane decomposition of grayscale image [11]

Discrete cosine transform

A discrete cosine transform (DCT) expresses a finite amount of input data as sum of cosine functions oscillating at different frequencies. These transformations have numerous applications in science and engineering. In image compression, the DCT-II introduced in 1974 by N. Ahmed, T. Natarajan and K. R. Rao [12] is used in practice. JPEG compression format uses it to convert content of the blocks to frequency domain. The transformation itself is not performing any form of data compression. Instead, it transforms the data to frequency domain and enables more effective quantization, therefore reduction of the output file size.

Discrete Wavelet transform

Just as DCT decomposes input signal into series of cosine functions, output of wavelet transform is set of wavelets which can be then effectively represented by their coefficients. For application in image compression, Cohen–Daubechies–Feauveau (CDF) wavelets [13] are used in practice in the WSQ and JPEG2000 compression algorithms. Similarly like DCT, DWT itself does not perform the compression, it rather enables more effective quantization.

2.2 Overview of formats

This section contains overview of the most widely used formats of image compression and comparison of the relevant ones with the WSQ.

BMP

The BMP is native bitmap format of Windows environment widely supported by other platforms too. Color model is palette based and supports multiple depths – from black and white up to 32-bit per pixel. Optionally, RLE lossless compression method can be used [14]. Disadvantage is relatively large file size due to optional low-ratio lossless compression method.

GIF

The Graphics Interchange Format (GIF) is raster format developed in 1987 [15]. It is widely supported even in legacy applications (as opposed to PNG). Color palette can be assigned of up to 256 colors from 24-bit RBG space. Animations are supported and widely used in practice. For compression, the LZW lossless method is used. Limited color palette makes GIF inadequate for complex images like digital photography.

PNG

The Portable Network Graphics (PNG) is raster format supporting lossless compression, widely used in the Internet environment. Both full-color RGB and palette-based color models are supported. The transparency is implemented by separate alpha channel or dedicated "transparent" color in both of the cases. The compression process consists of 2 stages [16]:

Filtering
: Filtering method is applied which predicts the value of next pixel based on the values of the surrounding ones. The difference from actual value is then used instead. This op-

eration makes following compression more effective since most of the values are close to 0.

Compression
Lossless compression method named DEFLATE is used. It is based on combination of LZ77 (predecessor of LZW) and Huffman coding.

Combination of these methods make PNG effective for representing images with large homogeneous areas without sacrificing visual quality. However, complex images with soft transitions usually result in large output file size.

JPEG

JPEG is one of the most popular and widely used compression formats for continuous-tone images. The compression process consists of several sequential steps [2, 17]:

Color space conversion
RGB color space is converted to YCbCr[5].

Chroma subsampling
Resolution of the chroma components (Cb, Cr) is reduced depending on the desired file size. This step benefits from the fact, that human eye is less sensitive to color details than luminance [3].

DCT computation
Image is subdivided into 8x8 pixel blocks. Discrete cosine transform (section 2.1) of the block is then computed. DCT transforms given block to linear combination of 64 patterns shown in figure 2.5.

Quantization
Frequency components are then quantized. Since human eye is more perceptive towards small differences in color

5. https://en.wikipedia.org/wiki/YCbCr

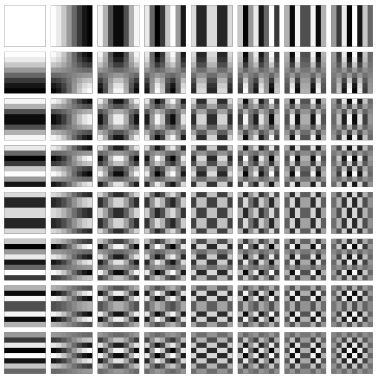

Figure 2.5: JPEG DCT frequencies [12]

and brightness in homogeneous areas [3], lower frequencies are quantized more accurately. The precision of quantization in general depends on the quality parameter.

Variable-length code assignment
Content of each block is then processed by Huffman coder.

JPEG is effective in compressing complex images with smooth transitions like digital photography where 10:1 compression ratio can be achieved with little to no perceptible loss in quality. However, on images with large homogeneous areas and sharp transitions, or when high compression ratio is desired, the individual blocks can be visible (see figure 2.6).

JPEG2000

JPEG2000 was developed in 2000 by Joint Photographic Experts Group committee in order to extend the functionality of JPEG and fix its shortcomings. Compression process consists of several steps [2]:

Color space conversion
Both irreversible (to YCbCr) and reversible (to YUV) transforms are possible.

Tiling
The image is split into rectangular regions called tiles,

which are then processed independently. This provides mechanism to access or edit part of the image, without the need to process all input data.

Wavelet transform
Both irreversible (CDF 9/7) and reversible (CDF 5/3) transforms are supported. The latter one does not introduce any quantization error and is therefore used for lossless compression. The transformation is applied on given tile and produces 4 subbands: horizontal, vertical, diagonal frequency characteristics and low resolution representation.

Quantization
Quantization in similar fashion like JPEG is performed. In case of lossless compression, this step is skipped.

Arithmetic coding
Coefficients of each tile's subbands are divided into blocks, which are individually coded by their bit-planes using the arithmetic coding.

The decoder is performing the inverse operations to all of the mentioned.

WSQ

The Wavelet Scalar Quantization is lossy compression algorithm specifically designed for 8-bit grayscale fingerprint images. It was developed by the FBI, the Los Alamos National Laboratory, and the National Institute of Standards and Technology [18]. Unlike the JPEG, image is not divided into blocks and the discrete wavelet transform (see section 2.1) is used instead of DCT. This eliminates block artifacts which are decreasing the amount of extractable features used for fingerprint comparison by disrupting its fine details. This can be seen in figure 2.6, which is showing comparison between the lossless compression (PNG 159 kB) and highly-compressed JPEG (11,3 kB) and WSQ (11.6 kB) images of the similar size. While block artifacts can be observed in JPEG, the perceived quality of WSQ image is objectively higher.

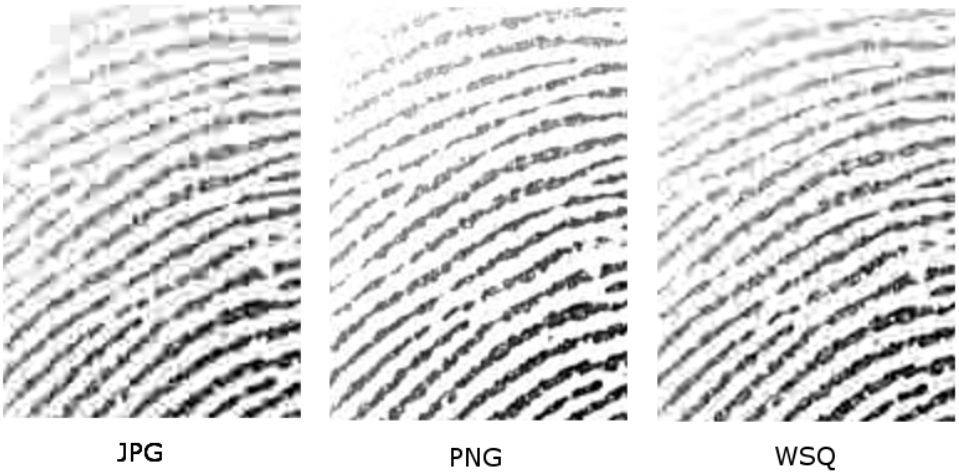

Figure 2.6: JPEG, PNG, WSQ comparison

The comparison with another algorithm using DWT – JPEG2000 was performed by NIST [19]. Results show little to no difference when using 500 ppi images, JPEG2000 is performing slightly better for higher resolutions. Further details about the WSQ are contained in chapter 3.

3 Wavelet Scalar Quantization

This chapter provides overview and description of every aspect of WSQ encoding process, description of data structures and algorithms used and structure of the output data. The content is based on the WSQ specification [20] and paper providing its brief overview [18].

Input image

The source fingerprint image is defined as grayscale 2-dimensional array of pixels with 256 intensity levels, digitized at resolution of 500 pixels/inch. Before the actual wavelet transformation of the image, raw input data is processed according to the equation [20]:

$$I'(m,n) = \frac{[I(m,n) - M]}{R} \tag{3.1}$$

$$0 \leq m \leq Y - 1$$
$$0 \leq n \leq X - 1$$

where X, Y are width/height and $I(m,n)$ denotes pixel value at given position in the image. The midpoint M and rescale R parameters are then part of the output data.

WSQ encoder in general

The algorithm can be split into three main subprocesses: the source fingerprint image is first decomposed using wavelet transform, scalar quantization is then applied to the wavelet coefficients and finally, quantizer indices are processed using lossless entropy coding. When applied to general image compression model defined in figure 2.1, basic structure matches as:

Transformation
: Two-dimensional symmetric wavelet transform (SWT) is used to decompose input to 64 sub-bands. These subbands contain floating-point wavelet coefficients.

Quantization
Result of the transformation is passed to bank of uniform scalar quantizers.

Symbol encoding
The integer indices output of quantization is then encoded by the run-length coding of zeros and Huffman coding.

Compressed output

Compressed data follow structure defined by special two-byte codes called markers. Some markers are then followed by the list of parameters creating the marker segment. Others are used only to indicate certain important position in the output like EOF[1]. Output file structure is described in detail in section 3.4. Depending on which segments are part of the output, 2 formats are defined by the specification [20]:

Interchange format
This format is meant to be used across different identification systems, therefore marker segments for filter coefficients and quantization and entropy-coding tables have to be included. This guarantees that decoding process has all the required information to successfully reconstruct compressed image.

Abbreviated format
This format is intended for use either within single system or applications where alternative ways of exchanging table-specification data exist. The structure is identical to the interchange format, however segments containing coefficients and tables are omitted, reducing the output size as result.

3.1 Subband decomposition

Subband decomposition is achieved by applying the symmetric wavelet transform, closer described in section 2.1. Wavelet transformation is

1. End of file

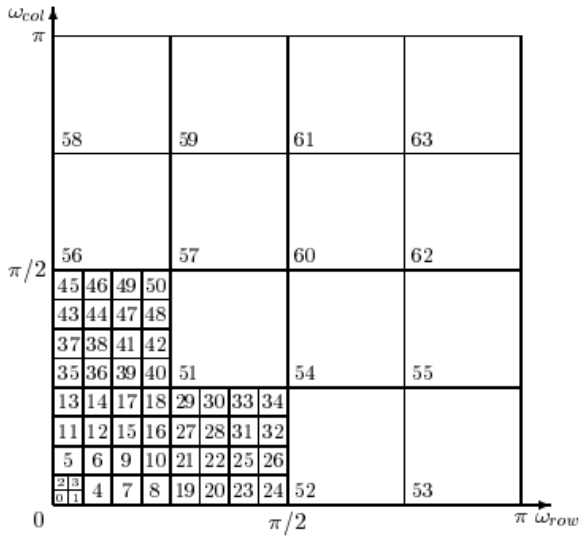

Figure 3.1: Subband decomposition [20]

applied first in the y-axis, then x-axis resulting in four-channel decomposition consisting of horizontal/vertical subband, diagonal subband and low-resolution subband. This process is recursively repeated until 64-band decomposition is achieved. Figure 3.1 displays the final frequency decomposition. It was specifically designed by analysis of spectral decomposition of information in fingerprint images.[21]

3.2 Quantization

The lossy compression itself is achieved by uniform scalar quantization of the SWT decomposition coefficients. This procedure maps analog floating-point wavelet coefficient to one of finitely many quantizer indices. Coefficients with low contribution to the final visual appearance are mapped more sparsely and therefore represented with lower amount of bits, while more significant ones are subjected to finer mapping. Since part of information is inevitably lost, this operation is irreversible and inherently lossy. Formula describing the quantization

encoding of subband is given by [20]:

$$p_k(m,n) = \begin{cases} \left\lfloor \frac{a_k(m,n) - Z_k/2}{Q_k} \right\rfloor + 1, & a_k(m,n) > Z_k/2 \\ 0, & -Z_k/2 \leq a_k(m,n) \leq Z_k/2 \\ \left\lceil \frac{a_k(m,n) + Z_k/2}{Q_k} \right\rceil - 1 & a_k(m,n) < -Z_k/2 \end{cases}$$

where $a_k(m,n)$ denotes two-dimensional subband, $p_k(m,n)$ is its quantization encoding, Z_k is the width of center quantization interval and Q_k width of quantization intervals in the k^{th} subband.

3.3 Entropy encoding

Output of the quantizer - indices $p_k(m,n)$ are then mapped to a stream of symbols contained in figure 3.2 and Huffman encoding is used to assign variable-length codes. This further reduces output data size by exploiting the non-uniform distribution of the symbols and assigning shorter codewords to more frequent symbols and vice versa (see Huffman coding description in section 2.1). The Huffman coding depends on the given image input and therefore its coding tables have to be part of the output file. Human tables are represented by the array of codeword lengths and a corresponding list of symbols. The exact method used for Human coding was adopted from the JPEG specification [17].

3.4 Structure of the data

This section contains overview of the WSQ file structure and its interpretation. Since all elements are represented with byte-aligned codes, the format consists of 8-bit blocks.

Semantics

Parameters

Parameters are integer values of 4-bit, 1-byte or 2-byte size, and characterize the encoding process, source image and other critical information without which the decoding process cannot properly reconstruct the image.

position	value
1	zero run length 1
2	zero run length 2
3	zero run length 3
.	
100	zero run length 100
101	esc for pos 8 bit coeff
102	esc for neg 8 bit coeff
103	esc for pos 16 bit coeff
104	esc for neg 16 bit coeff
105	esc for zero run - 8 bits
106	esc for zero run - 16 bits
107	coeff value 73
108	coeff value -72
109	coeff value -71
.	
180	- *use position 1 only* -
.	
253	coeff value 73
254	coeff value 74

Figure 3.2: Huffman table input symbols [20]

Markers
Markers are used to identify important parts of the compressed data structure. All markers are assigned one of the specific two-byte codes, for exact list see specification [20].

Marker segments
A marker segment consists of a marker followed by a sequence of related parameters. Its length is always defined by the first two-byte parameter.

Entropy-coded data segment (ECS)
These segments contain the actual entropy-coded image data of byte-aligned size to match the structure of the output format.

Restart marker (RST)
Restart marker is used to isolate ECS. Because they can be identified without the decoding process, they provide

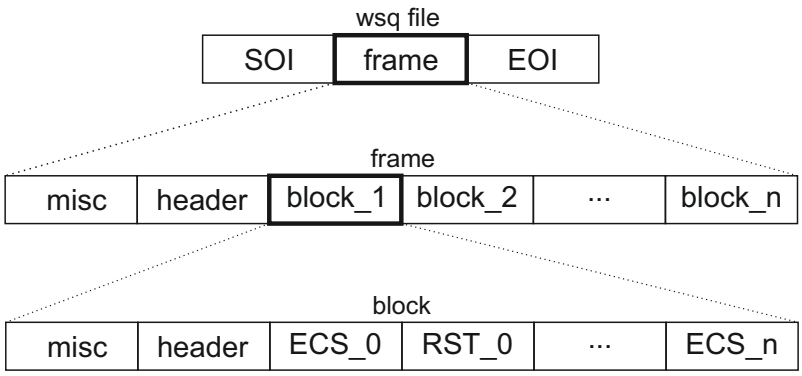

Figure 3.3: High-level syntax of .wsq file [20]

mechanism for progressive transmission and minimize data corruption.

Syntax

High-level structure and order of elements is specified in figure 3.3, their brief description follows:

SOI/EOI
Start of image/end of image markers

Frame header
Frame header is present at the start of a frame and specifies the source image characteristics and encoder version.

Block header
Block header is present at the start of a block segment and specifies the selection of the Huffman coding table used for all the subbands in the block.

Table/miscellaneous marker segment
Can contain either transform table-specification, quantization table-specification, huffman table-specification or comment.

For details including internal structure, ordering of the parameters and allowed values, see specification [20].

4 Home Credit Fingerprint Solution

This chapter describes architecture of the system where the WSQ encoder is used in practice – the FP_Solution, explanation of the Java Web Start technology and the limitations currently faced which led to demand for the custom WSQ encoder.

4.1 Architecture

In the context of the primary motivation of this implementation, it is important to describe the application where the encoder will be used in practice. FP_Solution is part of Home Credit's Homer Select Suite and is used for client identification in countries where traditional methods are not available or are not reliable enough. Its architecture is demonstrated in figure 4.1, consisting of three main components:

FP_Server
: Spring-based[1] Java application running on application server, stores fingerprint data sent from FP_Client, sends registration and identification requests to EAFIS (see below). Based on result and additional data about fingerprint set, evaluates and maintains list of information which further describe what does given match mean from the business perspective. These information are then reported to other company systems.

FP_Client
: Java application deployed on point of sale's computers. Performs extraction of the fingerprints, their validation and transmission to FP_Server. PF_Client is distributed by Java Web Start which is described in section 4.2.

EAFIS
: ExpressID AFIS[2] is biometric server providing fingerprint match results for the requests of FP_Server.

1. Java platform application framework
2. Automated Fingerprint Identification System

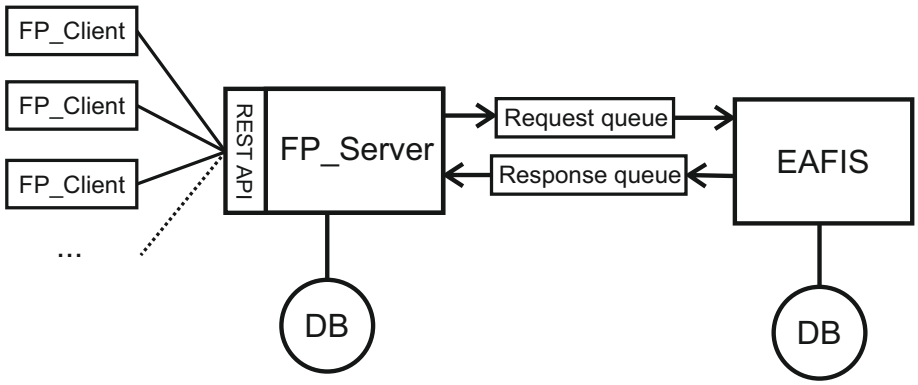

Figure 4.1: FP_Solution architecture

The workflow

In order to provide basic idea about the way how FP_Solution works and where does the WSQ encoding take place, description of client registration process is provided. It is focused only on FP_Solution and for sake of simplicity does not include other company systems which are part of the process.

1. FP_Client application is started and customer is logged in.
2. Once all required fingerprint data are collected, WSQ compression is performed and registration request containing the data is sent to FP_Server.
3. On FP_Server, validation is performed, data are saved to database and request is put to the request queue.
4. Once the response from EAFIS is in the response queue, FP_Server will process the found matches and save them as specific information to the database.

4.2 Java Web Start

Java Web Start is application deployment technology which allows launching of Java applications on client directly from web browser

[22]. When specific link is clicked, web server responds with `.jnlp` file which is then opened by the Java Web Start which runs the application according to the rules defined inside. Java Web Start is part of JRE[3] since release of Java 5.0.

Java Network Launching Protocol

JNLP is defined by XML Schema with following content: location of the `.jar`[4] files, required dependencies and rules defining the application launching mechanism. Application and resources are cached after downloading and checked for update on every subsequent launch. Generally, two update policies exist. Update can be required as prerequisite to actual start – this guarantees that the most recent version of application is always running. Second option is to allow immediate start and update during the run-time – here, the possibility of running multiple versions must be taken into account.

4.3 Limitations and possible solutions

While communication with fingerprint reader and FP_Server is performed by the FP_Client itself, for other critical functionality like template extraction, quality validation and WSQ compression, it has to rely on external biometric library – the IDKit[5] from Innovatrics. However, usage of this complex library is currently causing several problems:

Stability
: IDKit has dependency on OpenMP library which is containing bug occasionally causing FP_Client to fail. As mentioned in chapter 5.1, bugs in native code are very troublesome because application crashes without any feedback, which is confusing for the user and makes problem investigation difficult.

Portability
: The external dependencies of IDKit are restricting the amount

3. Java runtime environment
4. Java archive
5. https://www.innovatrics.com/idkit-fingerprint-sdk/

of available platforms. More importantly, these dependencies cannot be distributed via JNLP. They have to be installed manually, which makes installation process more complicated.

Footprint
The size of IDKit library is the issue, especially on computers with limited resources and connection bandwidth.

Innovatrics is also providing the light-weight alternative which has smaller footprint and no external dependencies, effectively solving all the previously mentioned problems. This comes at the cost of reduced functionality. FP_Client is using none of the affected methods, with the exception of WSQ compression support. Therefore, in order to replace the library, alternative WSQ encoder had to be used.

Alternatives

Several alternative WSQ implementations exist (Cognaxon[6], Smufs[7], Net-X[8]) but they are all proprietary and platform-dependent – introducing new native-code dependency was not desirable. Pure Java implementation by Lakota Software[9] was not acceptable because of pricing. Free solution written in Java, based on reference implementation already exists [23], but only decoder functionality is included. Therefore, the final decision was made to implement the encoder in similar fashion.

6. http://www.cognaxon.com/index.php?page=wsqlibrary
7. http://www.smufsbio.com/wsq-library.aspx
8. http://www.netxsolutions.co.uk/wsqlibrary.aspx
9. http://www.lakotasoftware.com/products#wsqparse

5 Implementation

This chapter contains description of the development process, differences between the C and Java programming languages, methods how to deal with them and structure of the implementation with characterization of the major classes.

5.1 The languages

The implementation is based on the reference implementation by NIST [1] written in C. In order to successfully transform it to Java, different principles and design trade-offs of these languages had to be assessed and the most fitting solution with intended usage of the code in mind had to be chosen.

C

Originally developed by Dennis Ritchie between 1969 and 1973 at Bell Laboratories [24], C is a general-purpose imperative computer programming language. It has a static type system, lexical variable scope and supports recursion. Programs are structured in `.c` source code files containing the implementation. The `.h` header files contain declarations and macro definitions and are exposing the interface to the corresponding `.c` file.

Compilation of C program is a multi step process consisting of [25]:

Preprocessing
: Before the actual compilation takes place, the source code file is transformed by the preprocessor. It is a separate program invoked automatically by compiler which interprets preprocessor directives reducing the repetitions in the source code. Typical usage is including files, defining macros, and conditional omission of parts of code.

Compilation
: In this stage, preprocessed code is translated to assembly instructions specific to the target processor architecture.

Some compilers skip the assembly step and produce the object code directly.

Assembly
An Assembler is used to translate assembly code to machine language instructions, also called the object code.

Linking
The final stage links object files to produce final executable file. The linker rearranges pieces of the object code and adds the missing parts so successful execution of the code is possible.

Java

Java is general-purpose object-oriented class-based programming language [26]. James Gosling, Mike Sheridan, and Patrick Naughton started the Java language project in 1991, first public implementation by Sun Microsystems was released in 1995. While language syntax is heavily influenced by C/C++, many low-level facilities are missing. There is a fundamental difference in compilation process between Java and C. Output of Java compiler is platform-neutral Java bytecode containing instructions for Java Virtual Machine. This means compiled Java code can run on all platforms supporting JVM without the need to recompile the code. JVM is an abstract computing machine that executes the bytecode.

Java Native Interface

Java Native Interface is a programming interface which allows Java code running in a Java Virtual Machine to operate with libraries and applications written in native code (C, C++ or Assembler) [27]. By using the JNI, native methods can create, update and inspect objects the same way Java code does, call Java methods, throw and catch exceptions, load classes and perform run-time type checking.

On the other side, it comes with some disadvantages. Debugging becomes more challenging since it has to be done using two debuggers simultaneously, one for Java and the other one for the native code. The advantage of platform-independence and portability is lost, with

partial workaround being providing separate version of native library for each of the intended platforms. Errors in the native code can also cause memory-access faults from which JVM cannot recover. As a result, JVM will immediately crash without any possibility of informing the user, capturing the information needed for investigation or cleaning up/releasing resources.

5.2 The differences

This section covers the practical differences between the languages which had to be dealt with during the development process. These characteristics rise out of specific intended use, focus on different aspects of software development and different date of origin of given languages.

Project structure

As mentioned in section 5.1, the C encoder groups all related method implementations to one `.c` file, corresponding `.h` file then contains constants, data structures definitions and function declarations.

Basic building blocks of Java project are `.java` files containing class, interface or enum definitions. Java files are organized into packages which provide them an unique namespace. Instead of the header file, API of the class is created by accessible class members (variables, methods, constants).

In case of WSQ encoder, all project files are contained in the same package, API is defined by the `Encoder` class described in section 5.3, rest of the classes remain hidden by the package-private access.

Preprocessor directives

The C preprocessor processes directives marked by the # (*sharp*) symbol and modifies the source code before the process of compilation. Generally, there are three uses of the preprocessing: including files, conditional compilation and macro definition and expansion [25].

Java has no preprocessor, so the same functionality had to be achieved by different means. While in C, external libraries are located by the linker at compile time (see section 5.1) (run-time for dynamic

libraries) and `#include` physically copies the content of the header file, Java's `Import` statement merely enables programmer to call a class by its unqualified name. The ClassLoader tries to locate it the first time its object is created or static member accessed. Conditional compilation was substituted by code refactoring and removal of unreachable parts. Macro definitions were replaced by static final class variables and static methods.

Memory management

Dynamic memory allocation allows the program to obtain more memory during the run time and release it once its not required. The important difference between the C and Java is in the process of deallocation. In C, dynamically allocated memory has to be freed explicitly. In Java, allocated memory associated with an object is freed automatically once its not used any more. This process is called garbage collection and is performed by the JVM [26]. Garbage collector regularly checks state of the objects and once no reference to given objects exist it is removed.

In the implementation, all code explicitly freeing memory was ignored. Instead, attention was paid not to leave references to unused objects in order to allow their automatic deallocation by the garbage collector.

Pointers

Pointers are powerful feature of C used to access the memory and manipulate the address. Pointer's value refers to value stored in memory using its address. Typical use cases of pointers in C include using pointer arithmetic to access arrays, using pointers as Strings, as writable function parameters and for performance optimization.

There is no explicit representation of C-like pointers in Java, the term "reference" is used instead. In Java, reference is value of reference type (class, interface, array and null). It is abstraction designed to hide its internal representation. Even though references are usually internally implemented by pointers (not required by specification [26]), there are several important differences.

The actual value of the reference (address in memory) is not accessible by the user, therefore no pointer arithmetics is possible. References are strongly typed, they can be cast to certain type only if the actual object referenced is of given type or its supertype. Java is strictly call-by-value, in case of both primitive and reference types, their value is passed as argument. Call-by-reference used in C can be simulated in Java on primitive types by using the wrapper class. In case of single argument, return value is used instead.

Arithmetics and data types

Special attention had to be paid to data type sizes. Data type size in C depends on target platform, only the lower bound is defined by specification [25]. Java's primitive data types are threated the same way across the platforms with the exception of floating-point arithmetics. Since JVM 1.2, float and double computations are not limited to 32/64-bit precision. Instead, on platforms which can handle other representations[1], higher precision will be used in intermediate computations to reduce round-off errors and overflows [26]. If strict portability and guaranteed identical result everywhere is required, `strictfp` modifier can be used to ensure all calculations are performed in identical precision across all platforms.

C allows to explicitly qualify char and integer-based data types as `unsigned`. This allows the bit previously used to define positive/negative sign to extend the maximum number the data type is able to hold. Prior to Java 8, unsigned data types were not supported, In Java 8 new methods supporting unsigned arithmetics have been added to the Integer and Long wrapper classes.

Another difference between the two is related to arrays and strings. C arrays can be seen just like syntactic sugar to access continuous memory spaces, Java wraps this representation with an object which holds its size for convenience. Strings in C are simply arrays of characters while in Java they are immutable objects. Java provides Unicode support by default, in C 3rd party library or custom implementation has to be used.

1. example: 80-bit extended precision on x86/x86-64

Last but not least, boolean has no explicit type in C, numerical representation where 0 stands for false and true is represented by every other value is used. Java provides both primitive type `boolean` and wrapper class `Boolean` which are using the `true` and `false` literals.

Error handling

Typical way of handling errors and non-standard situations in C is by returning the error code from function call where 0 denotes no issue. This error code is then mapped to the list of error descriptions. Java uses mechanism of exception handling where exception is an event which disrupts the normal flow of the program [26]. There are two types of exceptions: checked, handling of which is checked at compile-time and unchecked. An exception can be caught using the try-catch block where appropriate error handling can take place. Since Java 7 automatic resource management also called try-with-resources is available, which further simplifies manipulation with resources by automatically closing them when leaving the block (even in case of exception).

Naming conventions

One last aspect which has no impact on actual computation but rather code readability and maintainability is difference in naming conventions. C is using abbreviated names for file names, in case of function and variable names the underscore delimiter "_" is often used. Macros are by convention defined using upper case names, again delimited by underscore [25].

Java uses the same convention for constants definition. File names are matching the name of contained class/interface/enum which are using the `UpperCamelCase`. Variables and method names are using the `lowerCamelCase` [26].

5.3 Project structure

The implementation is structured as Apache Maven[2] project and follows the standard directory layout [28]. Unit tests are located under `src/test/java/` directory and rely on JUnit and AssertJ dependencies. Classes of the actual encoder are located under `src/main/java/` directory and here is short description of each one of them:

Constants.java
: Contains hi/lo-pass filter definitions and all constants used across the rest of the implementation.

DataStructures.java
: Contains all data structure definitions and methods working with them.

EncoderImpl.java
: Contains core functionality of WSQ algorithm such as creating internal data structures from raw input data, quantization and writing compressed data to output in defined format and structure.

Encoder.java
: Provides builder-based[3] API for user to set up the input parameters for EncoderImpl.

Description of input parameters of EncoderApi follows:

Width
: Width of the input image in pixels (mandatory).

Height
: Height of the input image in pixels (mandatory).

Quality
: Decimal number from interval <0.75, 2.2> which determines the quality of the output image (default value is 1.2).

2. A build automation tool used primarily for Java projects
3. Design pattern

PPI
: Resolution of the input image in pixels per inch. Is not actually used by the algorithm, only put to predefined section in the output file metadata (default value is 75).

Include metadata
: If set to false, the metadata at the beginning of the output file is omitted. This has no impact on decoding process (default value is true).

Commentary
: Adds provided commentary to predefined section in the output file metadata (default value is empty string).

6 Testing and evaluation

This chapter contains description of the implementation's testing process and achieved results. For each type of the test, motivation and methodology is provided. Unit tests are part of the project under the `src/test/java/` directory and focus on testing of encoder API by using various illegal and boundary inputs. Functional tests are used to verify that output of the encoder using certain parameters is matching its reference implementation counterpart. Non-functional testing of computational and memory footprint are performed to make sure FP_client application will run smoothly even on low performance machines.

6.1 Functional testing

Test verifying that for given input, encoder returns correct output is based on NIST certification procedure[1]. This procedure is used to verify that a custom implementation of WSQ encoder/decoder meets the WSQ specification. Fingerprint images of various resolutions are used as input in .raw format for encoder and .wsq for decoder.

Methodology

Since Java WSQ Encoder is based on reference implementation, the goal here is bit-to-bit identity between the outputs of these encoders. Each of 40 input images is encoded with both highest (2.25) and lowest (0.75) quality setting and output is then compared to the corresponding one of reference encoder.

6.2 Non-functional testing

This type of testing was performed to verify there was no excessive increase in computational and memory requirements which could render this implementation unusable, especially on low-performance computers as the ones used on the POS (see chapter 4).

1. https://www.nist.gov/programs-projects/wsq-certification-procedure

Computer	CPU (Intel)	Storage	RAM (GB)	OS
Workstation	i7-4600M	SSD	16	Windows 7
POS netbook	Atom D2500	HDD	2	Ubuntu 16.4

Table 6.1: Testing platforms

Methodology

Testing was performed on a notebook workstation used for development and netbook used at the point of sale – this is a typical hardware where FP_Client is usually running (see table 6.1 for additional information about the machines). As the input, two types of images were used: standard size produced by fingerprint reader used with FP_Solution and large 50 MP[2] to observe the algorithm's behavior with extreme pixel count. Further details about the images are provided in table 6.2.

Performance test was executed 5 times for each of the platforms, implementation and input data size. In case of standard-sized input, each time 10 separate images were processed in row and result was averaged to minimize system's interference. It was made sure that no intensive process was running on the background at the given time. Arithmetic mean and standard deviation of the results were computed and are part of the evaluation. In case of memory consumption, results were obtained using system's built-in process monitor for the reference implementation and NetBeans profiler[3] for the Java one in order to examine the structure of the allocated memory.

6.3 Evaluation

The functional test proven identity on binary level between the output of reference and Java encoder for both quality settings. The results in form of screenshot are available in figure 6.1.

Performance tests results are the content of tables 6.3, 6.4 and figure 6.2 for execution times, memory usage is available in table 6.5 and

2. megapixel – a million pixels
3. https://profiler.netbeans.org/

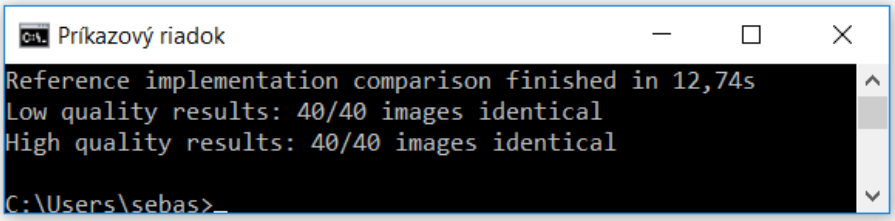

Figure 6.1: Encoder output comparison test

Input data	Count	Dimensions (WxH)	File size
standard	10	320x480	150 KB
large	1	4943x4240	19.9 MB

Table 6.2: Test images details

appendix contains structure of the memory allocated by the JVM in figures B.1 and B.2.

Execution times show, that the difference between the implementations is getting slightly lower when large input data are processed, independent of the platform. The important outcome is that 35.16% increase in processing-time of standard-sized data on netbook is acceptable and average computation-time of 363.6 ms will not bother the user of the application in any way.

Memory usage comparison is not so straightforward. Even though Java encoder process itself consumed significantly larger amount of memory than its reference counterpart – 8471.43% increase in case of standard-size input, these numbers have to be interpreted in regard to the differences in memory management between the languages (see section 5.2). Further analysis using the profiler (figure B.1) has shown the actual amount to be 13,1 MB, the rest was unused allocation by the JVM. Since the encoder is planned to be used inside the application already running in JVM, this increase is within expectations and acceptable.

Input data	Implementation	mean (ms)	st. deviation
standard	Reference	55.6	10.6
	Java	80.2	4.2
large	Reference	4034.0	60.8
	Java	5321.6	27.0

Table 6.3: Performance test - workstation

Input data	Implementation	mean (ms)	st. deviation
standard	Reference	269.0	9.7
	Java	363.6	13.0
large	Reference	9299.0	208.5
	Java	10762.2	427.3

Table 6.4: Performance test - netbook

Input data	Implementation	Memory allocation (MB)
standard	Reference	1.4
	Java	120.2 (used: 13.1)
large	Reference	180.3
	Java	442.5 (used: 352.6)

Table 6.5: Performance test - memory usage

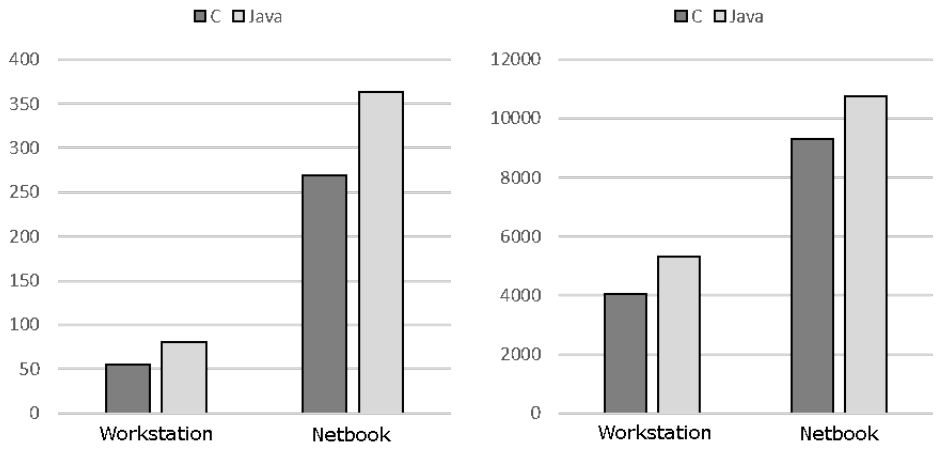

Figure 6.2: Average execution times (standard/large)

7 Conclusion

The primary goal of the thesis – implementation of WSQ encoder in Java was successfully accomplished and tested against the NIST reference implementation. Performance tests confirmed that there is no significant degradation of performance and this implementation is fully eligible for its intended application.

Current state

The WSQ Encoder was integrated into the new version of FP_Client which has been successfully tested by both system and user acceptance tests. After the testing was performed, it has been deployed to production at Home Credit branch in one of the countries. This brings benefits in form of higher stability, smaller footprint and wider portability into the business. The deployment to other countries is planned in the near future.

The future

One of the possible extensions for the future is combining this encoder with already existing Java WSQ decoder [23], implementing plugin for Java Image I/O and therefore creating pure Java WSQ codec[1]. In the context of FP_Solution, this could be used at the server side for adding support of WSQ decompression and it would allow to replace the native library the same way it has been done in FP_Client.

The other possibility is to apply for already mentioned NIST certification. However, since both the encoder and decoder are based on NIST reference implementation, it is clear their outputs conform to the WSQ specification.

1. encoder+decoder

Bibliography

1. *NIST Biometric Image Software (NBIS)* [online]. Gaithersburg: The National Institute of Standards and Technology, 2010–2016 [visited on 2017-04-04]. Available from: `https://www.nist.gov/services-resources/software/nist-biometric-image-software-nbis`.
2. GONZALEZ R.C.; WOODS, R.E. *Digital Image Processing*. 3rd ed. Pearson/Prentice Hall, 2008. ISBN 9780131687288.
3. KUNT, WINKLER; VAN DEN BRANDEN LAMBRECHT; *Vision and Video: Models and Applications*. 2nd ed. Springer, 2001. ISBN 9780792374220.
4. SHANNON, Claude E. A Mathematical Theory of Communication. *Bell Systems Technical Journal*. 1948.
5. HUFFMAN, D. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*. 1952.
6. VITTER, J. S. Design and Analysis of Dynamic Huffman Codes. *Journal of the ACM*. 1987.
7. KNUTH, Donald E. Dynamic Huffman Coding. *Journal of Algorithm*. 1985.
8. RISSANENISSA J.J.; LANGDON, G.G. Arithmetic Coding. *IBM Journal of Research and Development*. 1979.
9. DELP, DUDA; TAHBOUB; GADIL; The use of asymmetric numeral systems as an accurate replacement for Huffman coding. *Picture Coding Symposium*. 2015.
10. WELCH, T. A Technique for High-Performance Data Compression. *Computer*. 1984.
11. DAMATO, Alessio [online]. Wikimedia Commons [visited on 2017-05-04]. Available from: `https://upload.wikimedia.org/wikipedia/commons/4/48/Lichtenstein_bitplanes.png`.
12. [online]. Wikimedia Commons [visited on 2017-05-04]. Available from: `https://upload.wikimedia.org/wikipedia/commons/2/24/DCT-8x8.png`.
13. FEAUVEAU, COHEN; DAUBECHIES; Biorthogonal bases of compactly supported wavelets. *Communications on Pure and Applied Mathematics*. 1992.

14. *Microsoft Windows Bitmap File Format Summary* [online]. FileFormat.Info [visited on 2017-05-04]. Available from: `http://www.fileformat.info/format/bmp/egff.htm`.

15. *Graphics Interchange Format Specification* [online]. W3C [visited on 2017-05-04]. Available from: `https://www.w3.org/Graphics/GIF/spec-gif87.txt`.

16. *Portable Network Graphics Specification* [online]. W3C [visited on 2017-05-04]. Available from: `https://www.w3.org/TR/PNG/`.

17. *Digital Compression and Coding of Continuous-Tone Still Images*. 1991. Standard 10918-1 (a.k.a the JPEG standard).

18. BRADLEY Jonathan N.; BRISLAWN, Christopher M. The FBI Wavelet Scalar Quantization Standard for grayscale fingerprint image compression. *Los Alamos National Laboratory*. 1993.

19. GRANTHAM, LIBERT; ORANDI; Comparison of the WSQ and JPEG 2000 Image Compression Algorithms On 500 ppi Fingerprint Imagery. *National Institute of Standards and Technology* [online] [visited on 1993-04-03].

20. *WSQ GRAY-SCALE FINGERPRINT IMAGE COMPRESSION SPECIFICATION*. 2004. Version 3.1.

21. BRADLEY Jonathan N.; BRISLAWN, Christopher M. Compression of fingerprint data using the wavelet vector quantization image compression algorithm. *Los Alamos National Laboratory*. 1992.

22. *Java Web Start Technology* [online]. Oracle, 1993–2016 [visited on 2017-05-04]. Available from: `https://docs.oracle.com/javase/8/docs/technotes/guides/javaws/developersguide/overview.html#jws`.

23. *NIST Biometric Image Software (Java Implementation)* [online]. GitHub [visited on 2017-05-04]. Available from: `https://github.com/kareez/jnbis`.

24. RITCHIE, Dennis M. The Development of the C Language [online] [visited on 1993-04-03]. Available from: `http://www.bell-labs.com/usr/dmr/www/chist.html`.

25. KERNIGHAN Brian W.; RITCHIE, Dennis M. *The C Programming Language*. 2nd ed. Prentice-Hall, 1988. ISBN 0131103628.

26. *The Java Language Specification*. 2015. Java SE 8 Edition.

27. *Java Native Interface Specification* [online]. Oracle [visited on 2017-05-04]. Available from: `http://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html`.

28. *Maven Standard Directory Layout* [online]. The Apache Software Foundation [visited on 2017-05-04]. Available from: `https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html`.

A Abbreviations

WSQ	Wavelet scalar quantization
FBI	Federal Bureau of Investigation
NIST	National Institute of Standards and Technology
RLE	Run-length encoding
MSE	Mean square error
SNR	Signal-to-noise ratio
LZW	Lempel-Ziv-Welch
PNG	Portable Network Graphics
DCT	Discrete cosine transform
CDF	Cohen-Daubechies-Feauveau
SWT	Symmetric wavelet transform
EOF	End of file
EAFIS	ExpressID AFIS
AFIS	Automated Fingerprint Identification System
JWS	Java Web Start
JNLP	Java Network Launching Protocol
JNI	Java Native Interface
JVM	Java Virtual Machine
API	Application programming interface
PPI	Pixels per inch
POS	Point of sale

B Memory usage structure

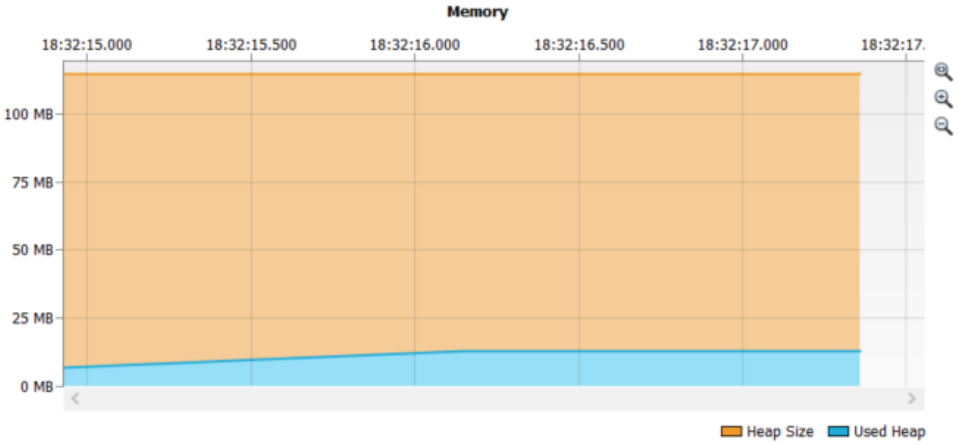

Figure B.1: Standard image profiler output

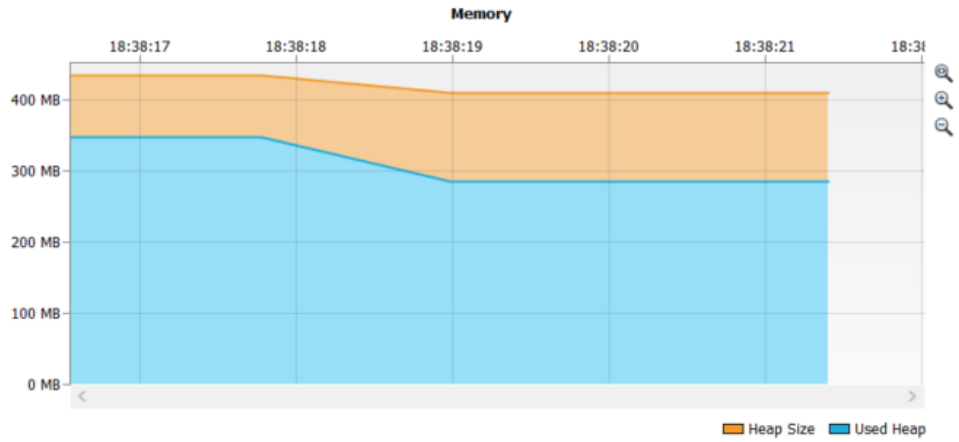

Figure B.2: Large image profiler output