Compression of RAW image files with the use of adaptive Huffman coding

KKO of 2020/2021

Author: **Patrik Németh**
Login: **xnemet04**

1 Overview

The output of this project is a program capable of compressing and decompressing RAW image files. The default compression method may be altered by the user via program options. By default, the images are compressed using run-length encoding, after which an adaptive Huffman tree is applied to the run-length encoded data. This results in a variable length coded compressed image. Two behaviour altering options were implemented: one for applying a subtraction model on the input image and one for adaptive image scanning (section 2.1). The compressed images are saved to a custom file format (figure 1). All of the compression and decompression is handled by the `Codec` class.

2 Image compression and representation

During compression, the image is first loaded from file by calling the `Codec::open_image()` method. This method is overloaded to be able to handle RAW images, as well as encoded ones. The RAW image is loaded via the `Image` class (section 4.1). Then the `Codec::encode()` method is called with the output file name and encoding options. Next, the subtraction model is applied, if needed, and the `Codec::rle()` method is called. If adaptive image scanning was requested by the user, then the best encoding direction is applied (see section 2.1). Last in the encoding process is the adaptive Huffman coding, which is handled by the `Huffman` class (section 4.2).

Before the encoded image data is written to file, some metadata is saved first. This metadata is 9 bytes wide and contains the image dimensions and encoding options. The compressed image file format is shown in figure 1. The encoding options have only 2 used bits, where the first 6 (most significant) bits are reserved and always set to zero, the 7-nth bit is set if vertical image scanning was used, and the last bit is set if the pixel subtraction model was used. The Huffman coded data directly follows the metadata. This data is to be read sequentially until EOF. The Huffman coded data always starts with a 0 bit.

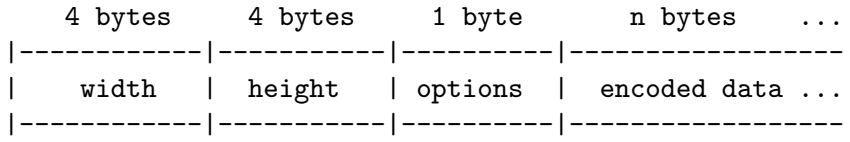

Figure 1: The file format for encoded images. Width and height are stored as big endian with relation to the start of the file. The encoded data is an adaptive Huffman code stream to be read sequentially from left to right.

2.1 Adaptive encoding direction

This was not implemented to specification due to time constraints. The `-a` program option changes the behaviour of the program, so that the best encoding direction for RLE (run-length encoding) is picked. The encoding directions are horizontal (same as default) or vertical. No per-block encoding is implemented.

The best encoding direction is picked by scanning over the input image in both directions and counting the number of times neighboring pixel values change. The scan direction with the fewer changes is the one that is better suited for RLE. The rationale is that if there are fewer value changes between neighboring pixels, then there are more frequent (and potentially longer) runs of same-value pixels.

3 Image decompression

The image is loaded by calling `Codec::open_image()`, which decodes the image. The decoding process is the inverse of the encoding process. This means that first, the metadata is read. Next, the Huffman coded data is loaded into a vector container for easier handling. After decoding the Huffman coded data, an inverse of RLE is applied to this data (in the correct direction based on the metadata). Next, if needed, the subtraction model is inverted and an instance of `Image` is created. After this, the image is written to file via `Codec::save_raw()`.

4 Implementation

This section contains brief information on the implementation of individual classes, datatypes or other aspects of the program implementation.

4.1 `Image` class

This is a simple class for basic image data representation. The image data is held in a vector container of type `uint8_t`, where each element represents a pixel value. The constructor is overloaded to be able to construct an instance based on RAW image data from a file or from a vector container. This class has methods for retrieving the image size, dimensions, has an overloaded indexing operator for direct access to pixel data. A method for writing raw pixel data to file was also implemented.

4.2 `Huffman` class

This class manages adaptive Huffman trees and handles all Huffman tree related operations. This class implements the FGK (Faller-Gallager-Knuth) algorithm. Upon instantiation, an initial tree is constructed, which is a single NYT node with no children. The Huffman tree is built specifically for pixel values (or any 8-bit symbols for that matter). However an additional value is expected to be inserted into the tree when no more values will follow, which is the EOF symbol defined in `Huffman.hpp` as `EOF_KEY`. This introduces an additional bit in the output code stream every time a new symbol is added to the tree (9 bits per symbol instead of 8). The EOF symbol is represented as value 256 and upon encountering this value during decoding, no more values are read from the input bit stream. The only public methods of this class are `Huffman::insert()`, `Huffman::decode()`, and `Huffman::reset_tree()`.

`Huffman::insert()` handles insertion of a symbol into the tree. This may be a new symbol or a symbol that is already in the tree. This method takes two parameters; the symbol to be inserted, and a pointer to a vector container of type `bool`. Upon symbol insertion, the Huffman code for the inserted symbol is appended to this vector.

`Huffman::decode()` takes two parameters; a pointer to a vector container of type `bool`, and a pointer to a vector container of type `uint8_t`. The first parameter must contain the bitstream to be decoded. The decoded data will be appended to the second parameter.

`Huffman::reset_tree()` simply resets the Huffman tree to its initial state.

4.3 `Codec` class

This class handles all the steps necessary for image encoding and image decoding. Its functionality was described in sections 2 and 3.

5 Usage

Before the application can be used, it needs to be compiled. An up-to-date `g++` compiler is required. To compile the program, use command `make` in the same directory as the source files. After compilation, the program can be launched with `./huff_codec {-c|-d} {-i in_file} {-o out_file} {-w width} [OPTIONS]`. Option `-c` tells the program to compress a RAW image stored in `in_file` and store it in `out_file`. The `-c` option requires the `-w` option, which tells the program how wide the RAW image is. Option `-d` tells the program to decompress an encoded image stored in `in_file` and store it in `out_file`. Additional options denoted by `[OPTIONS]` are:

- `-m` : use the subtraction model before encoding (has effect only with `-c`),
- `-a` : use adaptive scanning during encoding (has effect only with `-c`),
- `-h` : print help and exit.

6 Program evaluation

The following table shows some stastics for each of the provided RAW image files. All input files were of size 262144 bytes. The *Enc. size* column shows the encoded image size in bytes, *Enc. time* shows the time it took to encode the RAW image, *Dec. time* shows the time it took to decode the encoded image, and *bpc* shows bits per character, which shows how many bits were on average used in the encoded image per byte of the RAW image. The bpc value is calculated as $\text{bpc} = \frac{(\text{encoded file size}) * 8}{\text{original file size}}$.

No additional options					Option `-m`				
File	Enc. size	Enc. time	Dec. time	bpc	File	Enc. size	Enc. time	Dec. time	bpc
`df1h`	262571	3.111s	2.631s	8.01	`df1h`	650	0.042s	0.024s	0.01
`df1hvx`	80268	0.662s	0.452s	2.44	`df1hvx`	33523	0.300s	0.196s	1.02
`df1v`	3939	0.065s	0.058s	0.12	`df1v`	976	0.029s	0.020s	0.02
`hd01`	100391	1.170s	0.982s	3.06	`hd01`	88148	0.836s	0.666s	2.69
`hd02`	95577	1.098s	0.960s	2.91	`hd02`	86590	0.868s	0.694s	2.64
`hd07`	157742	1.556s	1.275s	4.81	`hd07`	109711	0.947s	0.632s	3.34
`hd08`	113813	1.051s	0.784s	3.47	`hd08`	98722	0.860s	0.643s	3.01
`hd09`	218002	2.265s	1.875s	6.65	`hd09`	152417	1.325s	0.975s	4.65
`hd12`	178036	1.918s	1.602s	5.43	`hd12`	126804	1.081s	0.783s	3.86
`nk01`	212591	1.947s	1.531s	6.48	`nk01`	198387	1.676s	1.336s	6.05
Avg	142293	1.484s	1.215s	4.33	**Avg**	89593	0.796s	0.596s	2.72
Option `-a`					Option `-m -a`				
File	Enc. size	Enc. time	Dec. time	bpc	File	Enc. size	Enc. time	Dec. time	bpc
`df1h`	3939	0.073s	0.058s	0.12	`df1h`	650	0.050s	0.021s	0.01
`df1hvx`	80268	0.640s	0.437s	2.44	`df1hvx`	33523	0.308s	0.189s	1.02
`df1v`	3939	0.071s	0.056s	0.12	`df1v`	976	0.053s	0.023s	0.02
`hd01`	100391	1.094s	0.978s	3.06	`hd01`	88148	0.891s	0.656s	2.69
`hd02`	95577	1.123s	0.965s	2.91	`hd02`	86590	0.877s	0.693s	2.64
`hd07`	157713	1.623s	1.316s	4.81	`hd07`	109450	0.881s	0.661s	3.34
`hd08`	112161	0.982s	0.740s	3.42	`hd08`	98667	0.869s	0.672s	3.01
`hd09`	216666	2.248s	1.843s	6.61	`hd09`	152214	1.250s	0.896s	4.64
`hd12`	176626	1.938s	1.606s	5.39	`hd12`	125944	1.112s	0.825s	3.84
`nk01`	212943	1.953s	1.539s	6.49	`nk01`	198694	1.716s	1.282s	6.06
Avg	116022	1.1745	0.9538	3.53	**Avg**	89486	0.8007	0.591s	2.72