A4: arith

Design Document
CSC 411 - Noah Daniels

Marceline Kelly

November 3, 2023

Contents

1 Overview

The goal of this assignment is to create a functional image compressor. The program contains the following modules:

- `rpeg`: Handles image compression and decompression.
- `bitpack`: Facilitates storing data in abstract words.
- `array2`: An intermediate collection format passed between operations.

The compression algorithm is composed of the following steps:

1. Read a PPM image.
2. Normalize each RGB pixel to a set of values between 0 and 1.
3. Convert the normalized image to component-video format.
4. Compress 2x2 blocks of component pixels using discrete cosine transformations.
5. Pack the compressed data into a series of 32-bit words.

   This operation makes extensive use of the `bitpack` crate.
6. Write the compressed image.

The steps are reversed for the decompression algorithm, with some minor alterations. Each step will have its own set of unit tests to ensure that all parts of the algorithm function correctly.

2 Architecture

This program is built upon the `Array2` data structure. Each operation takes in an `Array2` and produces a new `Array2`, except when directly reading from or writing to an output stream.

2.1 PPM modification

PPM images serve as both the input and output of this program. In order to work with the compression algorithm, all images must have an even width

and height. If the input image has an odd number of rows, we drop the bottommost row; the same applies for the rightmost column.

2.1.1 Functions

```
fn trim_image(img: Array2<Rgb>) -> Array2<Rgb>
```

Returns a copy of the image sans the last row and/or column if `height` and/or `width` is odd.

2.1.2 Test cases

- Each dimension of the output image should be equal to or one less than the respective dimension of the input image.
- An iterator over the input array minus the dropped elements is equivalent to an iterator over the output array. That is, both elements at a given position are identical.

2.2 Normalized RGB representation

A hallmark trait of the PPM format is the variable denominator (the maximum allowed brightness value). For consistency, we normalize this value to 1, followed by scaling each red, green, and blue sub-pixel in the image to the appropriate value between 0 and 1.

2.2.1 Types

`NormalizedRgb`

An RGB pixel where each value is a floating-point number between 0 and 1.

2.2.2 Functions

```
fn normalize_rgb(input: Rgb) -> NormalizedRgb
```

Scales each R, G, and B value between 0 and 1 based on its proximity to the denominator (represented as 1).

```
fn denormalize_rgb(input: NormalizedRgb) -> Rgb
```

Multiplies each normalized R, G, and B value by 255 then rounds to the nearest integer.

2.2.3 Test cases

- Normalization: given a denominator d, a `Rgb` $\{R\colon 0, G\colon \frac{d}{2}, B\colon d\}$ becomes a `NormalizedRgb` $\{R'\colon 0.0, G'\colon 0.5, B'\colon 1.0\}$, no matter the value of d.
- Denormalization: a `NormalizedRgb` $\{R\colon 0.0, G\colon 0.5, B\colon 1.0\}$ becomes `Rgb` $\{R'\colon 0, G'\colon 128, B'\colon 255\}$.
- All values in a `NormalizedRgb` fall between 0 and 1.
- All values in a denormalized `Rgb` fall between 0 and 255.

2.3 Component video representation

Our compression algorithm requires that images be stored in component video format (Y, P_B, P_R). To convert from the floating-point RGB values to component, we apply the following formula:

$$\begin{pmatrix} Y \\ P_B \\ P_R \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.168736 & -0.331264 & 0.5 \\ 0.5 & -0.418688 & -0.081312 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

Undoing this formula follows a similar process:

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1.0 & 0.0 & 1.402 \\ 1.0 & -0.344136 & -0.714136 \\ 1.0 & 1.772 & 0.0 \end{pmatrix} \begin{pmatrix} Y \\ P_B \\ P_R \end{pmatrix}$$

2.3.1 Types

`Component`

A component video pixel consisting of luma (Y) and chroma (P_B, P_R) floating-point values.

2.3.2 Functions

```
fn normal_rgb_to_component(input: NormalizedRgb) -> Component
```

Transforms a normalized RGB triple into a component triple.

```
fn component_to_normal_rgb(input: Component) -> NormalizedRgb
```

Transforms a component triple into a normalized RGB triple.

2.3.3 Test cases

- RGB to component: a `NormalizedRgb` produces a `Component` whose values match with known, expected results.
- Component to RGB: a `Component` produces a `NormalizedRgb` whose values match with known, expected results.
- Y falls between 0 and 1.
- P_B and P_R fall between -0.5 and 0.5.

2.4 Discrete cosine transformation

The actual compression of the image begins during this step. Given a 2x2 block of component values, we separate each luma value (Y_1, Y_2, Y_3, Y_4) then average the P_B ($\overline{P_B}$) and P_R ($\overline{P_R}$) values.

We then gather four coefficients, a, b, c, and d, using the following discrete cosine formulas:

$$\begin{aligned} a &= (Y_4 + Y_3 + Y_2 + Y_1)/4.0 \\ b &= (Y_4 + Y_3 - Y_2 - Y_1)/4.0 \\ c &= (Y_4 - Y_3 + Y_2 - Y_1)/4.0 \\ d &= (Y_4 - Y_3 - Y_2 + Y_1)/4.0 \end{aligned}$$

Finally, we produce a `CompressedBlock` that contains each of the four coefficients as well as the two average chroma values.

To return a compressed block of pixels to its original composite form, we first calculate the following values using inverse discrete cosine formulas:

$$\begin{aligned} Y_1 &= a - b - c + d \\ Y_2 &= a - b + c - d \\ Y_3 &= a + b - c - d \\ Y_4 &= a + b + c + d \end{aligned}$$

Then, we produce four composite pixels of the form (Y_i, $\overline{P_B}$, $\overline{P_R}$). Notice that each pixel uses the same pair of chroma values.

2.4.1 Types

`CompressedBlock`

A 2x2 block of pixels containing four floating-point luma coefficients (a, b, c, d) and two averaged floating-point chroma values ($\overline{P_B}$, $\overline{P_R}$).

2.4.2 Functions

```
fn compress_block(input: Array2<Component>) -> CompressedBlock
```

Transforms a 2x2 array of component pixels into a single set of chroma averages and luma coefficients.

```
fn decompress_block(input: CompressedBlock) -> Array2<Component>
```

Transforms a block of chroma averages and luma coefficients into a 2x2 array of component pixels.

2.4.3 Test cases

- The discrete cosine transformation produces proper results when tested against known values.
- a falls between 0 and 1.
- b, c, and d each fall between -0.5 and 0.5.
- The inverse discrete cosine transformation produces proper results when tested against known values.
- All P_B values within a decompressed set of component pixels are identical. Likewise with P_R.
- P_B and P_R fall between -0.5 and 0.5.
- All four Y values fall between 0 and 1.

2.5 Quantization & bit-packing

This part comprises the second compression step. To store each compressed block into a single 32-bit word, we must quantize some of the values. Since a always lies between 0 and 1, we can convert it into a 9-bit, unsigned integer by multiplying by 511 then rounding.

To quantize b, c, and d, we clamp each of the values between -0.3 and 0.3. Then, we multiply each value by 15 and round to the nearest integer to produce three signed integers between -15 and 15. Each of these can fit into 5 bits in our 32-bit word.

Finally, we convert the average chroma values into two unsigned index values using the `csc411_arith::index_of_chroma` function, each of which use 4 bits.

Using the `bitpack` crate, these values get stored into a 32-bit word in the following order:

(a x 9 bits, b x 5 bits, c x 5 bits, d x 5 bits, P_B x 4 bits, P_R x 4 bits)

At last, the compression is complete. Each of these words will be combined and output in our proprietary image compression format.

To decompress a 32-bit word, we simply perform the inverse calculations on the bit-packed a, b, c, and d integers. For the indexed chroma averages, we use the `chroma_of_index` function to undo the index calculation.

2.5.1 Functions

```
fn pack_word(input: CompressedBlock) -> u32
```

Packs the values of a compressed block of pixels into a single, 32-bit word.

```
fn unpack_word(input: u32) -> CompressedBlock
```

Unpacks a 32-bit word into a compressed block of pixels.

The functions in the `bitpack` crate are used within each of these functions but are not listed for brevity.

2.5.2 Test cases

- A left or right shift by 64 bits yields all zeroes.
- When `news` is given a `value` that does not fit in `width` signed bits, it returns `None`.
- When `newu` is given a `value` that does not fit in `width` unsigned bits, it returns `None`.

- For non-error values,
  `getu(newu(word, w, lsb, val).unwrap(), w, lsb) == val`.
- If `lsb2 >= w + lsb`, then
  `getu(newu(word, w, lsb, val), w2, lsb2) == getu(word, w2, lsb2)`
- The previous two tests apply to the signed variants, as well.

3 Information Loss

The following operations cause data loss within this compression algorithm:

1. Trimming extraneous rows and columns
   - Occurs only once
   - Insignificant for large images
2. Normalizing each RGB value
   - Occurs only once
   - The PPM denominator is set to 255 during decompression, leading to a loss in fidelity for images with a denominator >255
3. Averaging P_B and P_R
   - Occurs repeatedly
4. Converting luma coefficients to integers
   - Occurs repeatedly
   - Involves clamping b, c, and d as well as rounding to the nearest integer after multiplication
5. Converting chroma averages to integers
   - Occurs repeatedly