Connor Montague
Nick Robillard

A4 Design

Descriptions

Overall description: The arith project will allow users to compress and decompress images using a plethora of different modules that work together to form the desired end result. The three* main components of arith are: array2, rpeg (with its own subcomponents), and bitpack.

Module descriptions:

- **array2**: array2 will be used to represent data in a 2-Dimensional form. For this assignment, it will likely hold Rgb triplets and f32's (in separate instances)

- **rpeg**: The rpeg program will handle the vast majority of the algorithm which needs to be executed in order to compress/decompress the desired images.
    - **main.rs**: the main module, provided in the handout, will simply take input from the command line to determine if a file should be compressed or decompressed, and where the data will be stored after the operations are performed. It will make calls to the compress and decompress functions, which are defined elsewhere
    - **codec.rs**: codec will contain the behavior for the compression and decompression algorithms, however it will not contain the implementations of the specific functions that must be used to transform the image
    - **transforms.rs**: this module will contain the behavior for the functions that must be used in the compression/decompression steps. These steps are defined in class and on edstem as: trim excess rows/columns form the image, iteratively grab 2x2 pixel chunks of the image, turn the RGB u32 triplets into RGB f32 triplets using the image denominator, turn the RGB values into component video values (Y/Pb/Pr), and turn the component video values to Pr_{avg}, Pb_{avg}, a, b, c, d using the discrete cosine transform. Finally, using the behavior defined in the bitpack module, turn these values into a single u32, which will get packed into the lower 32 bits of a 64 bit word. It is important that all of these operations defined above have reverse versions.

- **bitpack**: The Bitpack module has multiple functions that make operations on individual bits of data. For example there is a function called fits that returns true if a signed value 'n' fits into 'width' signed bits. This module is designed to take bits and do operations on them to turn them into a single unsigned u32 int.

- **ppmdiff**: ppmdiff will be used to test the difference between two different .ppm files. It will achieve this by calculating the root mean square difference (RMSD) between the pixels in the two images, and return this value to standard output. It is important to note that ppmdiff will only work on images whose num rows/num cols differ by at most 1.

*ppmdiff is technically not a moving part when it comes to the execution of compression/decompression, but is an important facet of testing our algorithm.

Architecture

Overall: In the overall program, main will be able to communicate with and call the functions defined in codec.rs. Codec's functions will be able to receive parameters from some arbitrary source (main, in this case), and codec will be able to communicate with and call the functions defined in transforms.rs, without knowing their implementation. Codec will also be able to communicate with and call the functions defined in bitpack, without knowing their implementation either.

main.rs: No new functions defined here, but will call the functions 'compress' and 'decompress' defined in codec.rs

codec.rs:

- compress(filename: Option<&str>) → ()
    - This function will take an option containing either a filename or none, which will be passed to the 411 image crate to read it as an RGB image. Once the image is opened, the corresponding function calls for compression defined by codec.rs will be called. Once the compress ends, the compressed image data will be written back to standard out.
- decompress(filename: Option<&str>) → ()
    - This function will take an option containing either a filename or none, which will be passed to the 411 image crate to read it as an RGB image. Once the image is opened, the corresponding function calls for decompression defined by codec.rs will be called. Once decompressed, the resulting ppm image data will be written to standard out.

Connor Montague, Nick Robillard
A4: arith

transform.rs:

Miscellaneous:

- struct comp_vid {pb_av: f32, pr_av: f32, a: f32, b: f32, c: f32, d: f32}
    - Struct to hold the component video values because otherwise we're storing a 6 digit large tuple in our array2 and that would make us very sad
- new_cv(pb_av: f32, pr_av: f32, a: f32, b: f32, c: f32, d: f32) → comp_vid
    - Function to create a new component video struct

Compression steps:

- trim_img(img: &array2<RGB>) → array2<RGB>
    - (COMPRESSION) This function will take an array 2 and trim off the excess row/column if need be
- rbg_to_float(img: &array2<RGB>) → array2<(f32, f32, f32)>
    - (COMPRESSION) This function will take an array2 of RGB values from the given ppm image, and divide all these values by the image denominator, thus converting from an array2 of u16 RGBs to an array2 of f32 RGBs
- float_to_cv(img: &array2<(f32, f32, f32)> → array2<(f32, f32, f32)>
    - (COMPRESSION) This function will take an array2 of RGBs as f32s, and turn them to the Y/Pb/Pr values as defined by the instruction handout
- cv_to_dct(img:&array2<(f32, f32, f32)>) → array2<comp_vid>
    - (COMPRESSION) This function will take an array2 of Y/Pb/Pr values, and return an array2 holding a struct of the Pb/Pr averages, as well as a, b, c, d

Once an array2 of component video values are made, these values will be passed to the public functions defined by bitpack. Bitpack will return "words" (integer values) that will be fed into a vector of these values, which gets written to standard out as the compressed data

Decompression steps:

A set of "words" (integer values) will be read and unpacked by Bitpack, and these values will be turned into corresponding component video values in the comp_vid struct, which will be used to construct an array2 where all the mathematical operations will be reversed

- dct_to_cv(img:&array2<comp_vid>) → array2<(f32, f32, f32)>
    - (DECOMPRESSION) This function will take an array2 of structs holding the Pb/Pr averages and a, b, c, d and return an array2 holding the Y/Pb/Pr
- cv_to_float(img: &array2<(f32, f32, f32)>) → array2<(f32, f32, f32)>
    - (DECOMPRESSION) This function will take an array2 of component video values and turn them into the corresponding RGB values.

Connor Montague, Nick Robillard
A4: arith

- float_to_rgb(img: &array2<(f32, f32, f32)>) → array2<RGB>
    - (DECOMPRESSION) This function will take an array2 of f32 triplets and transform every triplet from f32s to u16s by multiplying each value by the image denominator

Connor Montague, Nick Robillard
A4: arith

bitpack.rs:
Directly from the assignment handout:

- pub fn fitss(n: i64, width: u64) → bool
  - Returns true iff the signed value `n` fits into `width` signed bits.
- pub fn fitsu(n: u64, width: u64) → bool
  - Returns true iff the unsigned value `n` fits into `width` unsigned bits.
- pub fn gets(word: u64, width: u64, lsb: u64) → i64
  - Retrieve a signed value from `word`, represented by `width` bits beginning at least-significant bit `lsb`.
- pub fn getu(word: u64, width: u64, lsb: u64) → u64
  - Retrieve an unsigned value from `word`, represented by `width` bits beginning at least-significant bit `lsb`.
- pub fn news(word: u64, width: u64, lsb: u64, value: i64) → Option<u64>
  - Return a modified version of the unsigned `word`, which has been updated so that the `width` bits beginning at least-significant bit `lsb` now contain the unsigned `value`. Returns an `Option` which will be None iff the value does not fit in `width` unsigned bits
- pub fn newu(word: u64, width: u64, lsb: u64, value: u64) → Option<u64>
  - Return a modified version of the unsigned `word`, which has been updated so that the `width` bits beginning at least-significant bit `lsb` now contain the signed `value`. Returns an `Option` which will be None iff the value does not fit in `width` signed bits.

Connor Montague, Nick Robillard
A4: arith

Testing

In order to test our compressor, we will do round trip testing on each component of the compressor using the ppmdiff project we're writing as a part of Lab 6. For the first few steps, very little data should be lost since we are only converting from u16 to f32. However, for the later steps, more data is expected to be lost (see below). However, if ppmdiff shows that an erroneous amount of data is lost at a certain step then that will help us narrow down potential sources of error.

In order to test bitpack specifically, we will use the testing module written as part of lab 5 in order to verify that certain mathematical properties hold true when performing the operations defined in the bitpack module. As described in the handout, it is sensible that we will exhaustively test the algebraic laws for an initial width and least significant bit, but will have to randomly test on values for a secondary width and least significant bit.

Additionally, we will have to test the individual methods contained within bitpack to verify their functionality. For example, we will have to write tests to verify the output returned by the fitss/fitsu functions when given a value that will/won't fit inside of [width] signed/unsigned bits. We will also need to make sure that errors are thrown when certain invariants are violated, such as $0 \leq w \leq 64$ and $w + lsb \leq 64$, as defined in the assignment/lab 5 handouts.

An image is compressed and then decompressed. Identify all the places where information could be lost. Then it's compressed and decompressed again. Could more information be lost? How?

When an image is compressed for the first time, a potential source of information loss is any odd row/column, since this will be trimmed. Additionally during compression, b, c and d get converted into 5-bit signed values, but a major assumption is that they lie between -0.3 and 0.3, when in reality they could lie between -0.5 and 0.5. As such, in order to fit the data into a more precise form, these rarer cases get thrown away and are thus lost during compression.

When an image gets decompressed, a major loss of information is the Pb and Pr averages. You can't "unaverage" a number back into its original values, so instead the average of the 4 Pb/Pr values will simply be copied into these values, which could be a major loss of information depending on the standard deviation of these Pb/Pr values. A more minor source of information loss when decompressing is converting from f32 RGB to u16 RGB, because on the chance that the f32s have decimal values these will be lost when turning into integers.