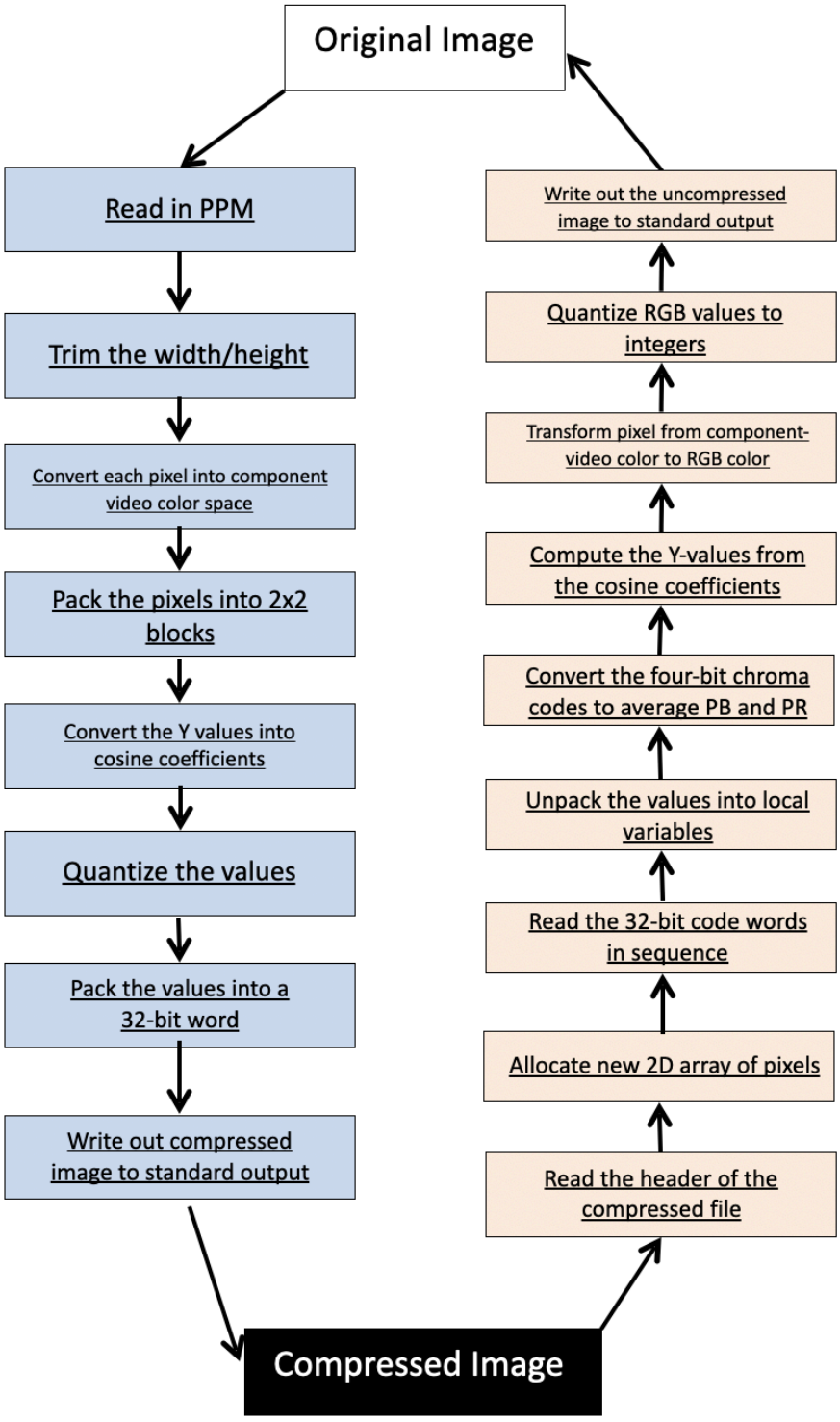
Madeline Lei, Diana Calderon
mlei03, dcalde02



Compression Implementation Plan

Step 1: Read in PPM

Description: Use the Pnm_ppmread() function from pnm.h to read in the PPM image.
Inputs: PPM file stored in a file pointer or stdin
Outputs: A Pnm_ppm struct representing the PPM image
Information loss: None
Testing: Print out the new Pnm_ppm struct to make sure its the correct image and format

Step 2: Trim the width/height

Description: If the PPM's width or height is odd, trim the last column or row to make both of the dimensions even
Inputs: a Pnm_ppm struct
Outputs: a potentially trimmed Pnm_ppm struct with even height and width
Information loss: Yes because if the PPM has odd dimensions, the information in the last row/column will be lost
Testing: Provide a PPM with an odd width, another with an odd height, and another with both an odd width and height, and print the dimensions to make sure they are even and the ones that needed to be trimmed were. Also print for a PPM with already even dimensions.

Step 3: Convert each pixel into component video color space

Description: Convert each pixel to a float. Then, transform each pixel from the RGB color space into component video color space (Y, P_B, P_R) using the formulas given in the spec.
Inputs: an array of Pnm_rgb pixels
Outputs: an array of floats representing those pixels in component video color space
Information loss: None
Testing: Choose pixels from an image and do the calculations on paper to make sure they match up using unit testing

Step 4: Pack the pixels into 2x2 blocks

Description: For each 2x2 block in the array, find the average of the P_B and P_R values of the four pixels. Then, convert these averages into four bit values using the `Arith40_index_of_chroma` function that is provided by the spec.
Inputs: an array of floats representing the pixels of the PPM in component video color space
Outputs: an array of unsigned integers representing 4-bit quantized representations of these values
Information loss: Yes, since we are finding the average of the values, we will lose the original values
Testing: Calculate the averages and unit test to make sure it matches that of the function. Also make sure the size is correct.

Step 5: Convert the Y values into cosine coefficients

Description: Use a DCT, transform the four Y-values of the pixels into cosine coefficients a, b, c, d.
Inputs: 4 floats representing the Y-values of the pixels
Outputs: 4 floats representing the cosine coefficients
Information loss: None
Testing: Choose pixels from an image and do the calculations on paper to make sure they match up using unit testing

Step 6: Quantize the values

Description: Code `a` into a 9-bit unsigned value. Convert `b`, `c`, and `d` into five-bit signed values (assuming they lay between -0.3 and 0.3). Finally, quantize P_R and P_B into 4-bit unsigned indices.
Inputs: An array of floats representing the image and chroma data.
Outputs: The index of the quantized value in an internal table
Information loss: Yes, we are converting a bunch of values in a range into one number, which may result in information loss.
Testing: Choose some pixels and make sure they quantize to the correct values using unit testing. Also make sure the sizes and types are correct.

Step 7: Pack the values into a 32-bit word

Description: Using the Bitpack interface, pack the quantized values into a 32-bit word
Inputs: An array of the quantized values
Outputs: A 32-bit word
Information loss: None
Testing: Make sure the types and sizes are correct. The next step also tests this.

Step 8: Write out compressed image to standard output

Description: Using printf, write the header of the compressed binary image, followed by a newline and then a sequence of 32-bit code words, one for each 2x2 block of pixels, with width and height variables describing the trimmed off images.
Inputs: The array of 32-bit words
Outputs: Compressed image to standard output
Information loss: None
Testing: Use the function we created in lab ppmdiff to compare the original and compressed images, making sure they are within the range of similarity

Decompression Implementation Plan

Step 1: Read the header of the compressed file

Description: Use `fscanf` to read in the width and height of the PPM image

Inputs: PPM image from a file pointer or standard input
Outputs: None, but store the width and height in a struct
Information loss: None
Testing: Provide files with no height or width to ensure it will CRE, and also provide it a file from stdin or as an argument. Make sure the heights and widths are correct.

Step 2: Allocate new 2D array of pixels

Description: Allocate 2D array of pixels of given width and height, with a size parameter being the size of the colored pixel. Place array, width, height and denominator in a Pnm_ppm struct.
Inputs: Width and height struct
Outputs: A Pnm_ppm struct representing the compressed image
Information loss: None
Testing: Print out the array to make sure it has the correct width, height, and denominator

Step 3: Read the 32-bit code words in sequence

Description: Read the 32-bit code words into the 2D array from the last step
Inputs: A file stream containing the compressed PPM. Will CRE if supplied file is too short
Outputs: A 2D array of 32-bit words containing all the image data
Information loss: None
Testing: Provide a file that is too short (the number of codewords is too low for the stated width and height, or the last one is incomplete) as well as some with proper 32-bit code words

Step 4: Unpack the values into local variables

Description: Using the Bitpack interface, unpack the a, b, c, d, and the coded average P_R and P_B values and store them in local variables
Inputs: The 2D array of 32-bit words containing all the image data
Outputs: The new values, stored as floats and 4-bit unsigned indices
Information loss: None
Testing: Print out the values from a known compressed image, and make sure they are the same as the original, using unit testing

Step 5: Convert the four-bit chroma codes to average PB and PR

Description: Use the Arith40_chroma_of_index function provided to convert the 4-bit coded chroma codes into the average P_R and P_B
Inputs: The 4-bit coded chroma codes as unsigned indices
Outputs: The average P_R and P_B as floats
Information loss: None
Testing: Print out the values from a known compressed image, and make sure they are the same as the original, using unit testing

Step 6: Compute the Y-values from the cosine coefficients

Description: Use the inverse of the discrete cosine transform to compute Y1, Y2, Y3, and Y4 from a, b, c, and d.
Inputs: The cosine coefficients, as unsigned and signed integers
Outputs: Floats representing the Y-values
Information loss: None
Testing: Print out the values from a known compressed image, and make sure they are the same as the original, using unit testing

Step 7: Transform pixel from component-video color to RGB color

Description: Use the formulas given in the spec, compute the original RGB color data from the values in the component video color space
Inputs: The average P_R and P_B as floats
Outputs: Transformed pixels as RGB color as floats
Information loss: None
Testing: Print out the values from a known compressed image, and make sure they are the same as the original, using unit testing

Step 8: Quantize RGB values to integers

Description: Quantize the RGB values to integers into a range that is appropriate for the denominator
Inputs: The RGB values as floats
Outputs: The RGB values as integers
Information loss: Yes, there is a loss of information during quantization since we are converting values in a range to one number
Testing: Print out the values from a known compressed image, and make sure they are the same as the original, using unit testing

Step 9: Write out the uncompressed image to standard output

Description: Using Pnm_ppmwrite(), write out the uncompressed image to standard output
Inputs: A pnm_ppm struct representing the uncompressed image
Outputs: None
Information loss: None
Testing: Use the function we created in lab ppmdiff to compare the compressed and decompressed images, making sure they are within the range of similarity