CS 4481b Assignment 3 Report

Paul Bartlett

Peppers Offset Histograms

Goldhill Offset Histograms

Comments: The images with larger offsets show the same impulse pattern for both images. This is because the width of the Goldhill and Peppers images are 360 and 256 respectively, and there is a greater chance of matching a pixel in the same proximity as the original pixel. This causes the frequency of matches to increase every time the width of the image is reached. This can be clearly seen in the smaller offsets where the peppers frequency is greatest at the two ends since the width of the image is equal to the offset. Similarly, this can’t be seen with the Goldhill 256 histogram because the width of the image is greater than the offset.

Peppers Matching Length Histograms

Goldhill Matching Length Histograms

Comments: The matching length histograms look fairly similar when using a log scale, but the size of match length 0 specifically decreases by a great amount when the matching length is increased. This is of course because the program scans over more values and is able to match with a previous value in the buffer easier. The Goldhill images have more outliers with larger matching length, but the values for each frequency are fairly consistent between both images for each respective matching length.

Average, Standard Deviation, and encoding/decoding time

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Offset Ave. | Offset Std. | Match Ave. | Match Std. | Encoding Time (s) | Decoding Time (s) |
| Peppers 256 | 73.32 | 90.35 | 0.85 | 0.57 | 0.04 | 0.01 |
| Peppers 1024 | 239.44 | 258.25 | 1.22 | 0.56 | 0.10 | 0.01 |
| Peppers 5120 | 1038.01 | 1301.39 | 1.57 | 0.66 | 0.39 | 0.01 |
| Goldhill 256 | 45.36 | 65.99 | 0.84 | 0.76 | 0.06 | 0.02 |
| Goldhill 1024 | 213.22 | 242.39 | 1.27 | 0.81 | 0.16 | 0.02 |
| Goldhill 5120 | 1044.66 | 1314.11 | 1.64 | 0.93 | 0.61 | 0.02 |

Justification for suitable searching\_buffer\_size

For both these images I would say that the searching\_buffer\_size of 1024 was the most suitable. When looking at the histograms of the offsets for 5120, once the offset reaches around 1000 the impulses stay around the same. Although it is good to still have matches, the offset numbers themselves are much too high when considering that the match length not frequently above 2. The improvement in match length average is greater for the Goldhill image than the peppers image, but it comes at the cost of a significantly higher encoding time. Even though the searching\_buffer\_size of 5120 improves the match length, it doesn’t create enough improvement to justify using it when 1024 offers decent encoding at a much faster rate.

Programs

lz77\_encoding\_function.c

#include "lz77\_encoding\_function.h"

void Encode\_Using\_LZ77(char \*in\_PGM\_filename\_Ptr, unsigned int searching\_buffer\_size, float \*avg\_offset\_Ptr, float \*std\_offset\_Ptr, float \*avg\_length\_Ptr, float \*std\_length\_Ptr) {

// Image variables

struct PGM\_Image pic\_pgm;

load\_PGM\_Image(&pic\_pgm, in\_PGM\_filename\_Ptr);

int \*pixel\_array = malloc(pic\_pgm.height \* pic\_pgm.width \* sizeof(int)), pixel\_count = 0;

// Iterate through all pixels to flatten array

for(int row = 0; row < pic\_pgm.height; row++) {

for(int col = 0; col < pic\_pgm.width; col++) {

pixel\_array[pixel\_count] = pic\_pgm.image[row][col];

pixel\_count++;

}

}

// LZ77 token arrays

int \*offset = calloc(pixel\_count, sizeof(int)),

\*matching\_length = calloc(pixel\_count, sizeof(int)),

\*next\_symbol = calloc(pixel\_count, sizeof(int));

int current\_buffer\_size = 0, matches, tok;

// Iterate through all pixels to create all LZ77 tokens

for(tok = 0; current\_buffer\_size < pixel\_count; tok++) {

// Buffer through previous values and search for matches

for(int i = 1; i <= searching\_buffer\_size && i <= current\_buffer\_size; i++) {

matches = 0;

// Continue looping for more matches

while(pixel\_array[current\_buffer\_size + matches] == pixel\_array[current\_buffer\_size - i + matches] && current\_buffer\_size + matches < pixel\_count) {

matches++;

}

// Update match length only if larger

if(matches > matching\_length[tok]) {

matching\_length[tok] = matches;

offset[tok] = i;

}

}

// Include matched characters when determining next symbol, then add that one to buffer size

current\_buffer\_size += matching\_length[tok];

next\_symbol[tok] = pixel\_array[current\_buffer\_size];

current\_buffer\_size++;

}

// Create files and file names

int buffer\_string\_size = (int)(ceil(log10(searching\_buffer\_size))+1);

char lzFileName[strlen(in\_PGM\_filename\_Ptr) + buffer\_string\_size + 5];

char offsetsFileName[strlen(in\_PGM\_filename\_Ptr) + buffer\_string\_size + 14];

char lengthsFileName[strlen(in\_PGM\_filename\_Ptr) + buffer\_string\_size + 14];

sprintf(lzFileName, "%s.%d.lz", in\_PGM\_filename\_Ptr, searching\_buffer\_size);

sprintf(offsetsFileName, "%s.%d.offsets.csv", in\_PGM\_filename\_Ptr, searching\_buffer\_size);

sprintf(lengthsFileName, "%s.%d.lengths.csv", in\_PGM\_filename\_Ptr, searching\_buffer\_size);

FILE \*lzFilePointer = fopen(lzFileName, "wb");

if(lzFilePointer == NULL) printf("Error opening lz file for writing");

FILE \*offsetsFilePointer = fopen(offsetsFileName, "wb");

if(lzFilePointer == NULL) printf("Error opening offsets csv file for writing");

FILE \*lengthsFilePointer = fopen(lengthsFileName, "wb");

if(lzFilePointer == NULL) printf("Error opening lengths csv file for writing");

// Write the lz header

fprintf(lzFilePointer, "P2\n%d %d\n%d\n%d %d\n", pic\_pgm.width, pic\_pgm.height, pic\_pgm.maxGrayValue, searching\_buffer\_size, tok);

int \*offset\_frequency = calloc(tok, sizeof(int));

int \*length\_frequency = calloc(tok, sizeof(int));

int offset\_sum = 0, length\_sum = 0;

// Write the LZ77 arrays for offsets, matching lengths, and next symbols (and csv data)

for(int i = 0; i < tok; i++) {

fprintf(lzFilePointer, "%d ", offset[i]);

offset\_frequency[offset[i]]++;

}

for(int i = 0; i < tok; i++) {

fprintf(lzFilePointer, "%d ", matching\_length[i]);

length\_frequency[matching\_length[i]]++;

}

for(int i = 0; i < tok; i++)

fprintf(lzFilePointer, "%d ", next\_symbol[i]);

// CSV data

for(int i = 0; i < tok; i++) {

if(offset\_frequency[i] != 0) {

fprintf(offsetsFilePointer, "%d,%d\n", i, offset\_frequency[i]);

}

if(length\_frequency[i] != 0) {

fprintf(lengthsFilePointer, "%d,%d\n", i, length\_frequency[i]);

}

offset\_sum += offset[i]; // for mean and stdev

length\_sum += matching\_length[i]; // for mean and stdev

}

fclose(lzFilePointer);

fclose(offsetsFilePointer);

fclose(lengthsFilePointer);

// Calculate the average and standard deviation of the offsets and match lengths

float offset\_mean, offset\_stdev = 0.0, length\_mean, length\_stdev = 0.0;

offset\_mean = (float) offset\_sum / tok;

length\_mean = (float) length\_sum / tok;

for(int i = 0; i < tok; i++) {

offset\_stdev += pow(offset[i] - offset\_mean, 2);

length\_stdev += pow(matching\_length[i] - length\_mean, 2);

}

offset\_stdev = sqrt(offset\_stdev / tok);

length\_stdev = sqrt(length\_stdev / tok);

// Save values in function parameters

\*avg\_offset\_Ptr = offset\_mean;

\*avg\_length\_Ptr = length\_mean;

\*std\_offset\_Ptr = offset\_stdev;

\*std\_length\_Ptr = length\_stdev;

// Free memory

free\_PGM\_Image(&pic\_pgm);

free(pixel\_array);

free(offset);

free(matching\_length);

free(next\_symbol);

free(offset\_frequency);

free(length\_frequency);

}

lz77\_encoding\_function.h

#ifndef LZ77\_ENCODING\_FUNCTION\_H

#define LZ77\_ENCODING\_FUNCTION\_H

#include <stdio.h>

#include <string.h>

#include <math.h>

#include "libpnm.h"

void Encode\_Using\_LZ77(char \*in\_PGM\_filename\_Ptr, unsigned int searching\_buffer\_size, float \*avg\_offset\_Ptr, float \*std\_offset\_Ptr, float \*avg\_length\_Ptr, float \*std\_length\_Ptr);

#endif // LZ77\_ENCODING\_FUNCTION\_H

lz77\_decoding\_function.c

#include "lz77\_decoding\_function.h"

void Decode\_Using\_LZ77(char \*in\_compressed\_filename\_Ptr) {

char c;

int row, col, width, height, maxGrayValue, searching\_buffer\_size, current\_buffer\_size = 0, pixel\_count, tokens;

struct PGM\_Image pgmImage;

// Open file for reading

FILE \*lzFilePointer = fopen(in\_compressed\_filename\_Ptr, "rb");

if(lzFilePointer == NULL) printf("Error opening lz file for reading");

// Make sure the first char is P

if(fgetc(lzFilePointer) != 'P') {

printf("Invalid PGM image: missing P");

fclose(lzFilePointer);

}

// Make sure the second char is either a 2 or 5

c = fgetc(lzFilePointer);

if(c != '2' && c != '5') {

printf("Invalid PGM image: missing 2 or 5");

fclose(lzFilePointer);

}

// Get the width, height, max gray value, and searching buffer size of the image

width = geti(lzFilePointer);

height = geti(lzFilePointer);

maxGrayValue = geti(lzFilePointer);

searching\_buffer\_size = geti(lzFilePointer);

tokens = geti(lzFilePointer);

create\_PGM\_Image(&pgmImage, width, height, maxGrayValue);

// Get all values from the offset, matching lengths, and next symbol arrays

pixel\_count = width \* height;

int \*offset = calloc(tokens, sizeof(int)),

\*matching\_length = calloc(tokens, sizeof(int)),

\*next\_symbol = calloc(tokens, sizeof(int)),

\*pixel\_array = malloc(pixel\_count \* sizeof(int));

for(int i = 0; i < tokens; i++)

offset[i] = geti(lzFilePointer);

for(int i = 0; i < tokens; i++)

matching\_length[i] = geti(lzFilePointer);

for(int i = 0; i < tokens; i++)

next\_symbol[i] = geti(lzFilePointer);

fclose(lzFilePointer);

// Decode the data from the arrays

for(int tok = 0; tok < tokens; tok++) {

if(matching\_length[tok] > searching\_buffer\_size)

printf("Matching length greater than buffer\n");

// Add matching length number of items

for(int i = 0; i < matching\_length[tok]; i++) {

pixel\_array[current\_buffer\_size] = pixel\_array[current\_buffer\_size - offset[tok]];

current\_buffer\_size++;

}

pixel\_array[current\_buffer\_size] = next\_symbol[tok];

current\_buffer\_size++;

}

pixel\_count = 0;

// Fill image with decoded values

for(row = 0; row < height; row++) {

for(col = 0; col < width; col++) {

pgmImage.image[row][col] = pixel\_array[pixel\_count];

pixel\_count++;

}

}

// Save image

char lzFileName[strlen(in\_compressed\_filename\_Ptr) + 5];

sprintf(lzFileName, "%s.pgm", in\_compressed\_filename\_Ptr);

save\_PGM\_Image(&pgmImage, lzFileName, 0);

// Free memory

free\_PGM\_Image(&pgmImage);

free(offset);

free(matching\_length);

free(next\_symbol);

free(pixel\_array);

}

lz\_decoding\_function.h

#ifndef LZ77\_DECODING\_FUNCTION\_H

#define LZ77\_DECODING\_FUNCTION\_H

#include <stdio.h>

#include <string.h>

#include "libpnm.h"

void Decode\_Using\_LZ77(char \*in\_compressed\_filename\_Ptr);

#endif // LZ77\_DECODING\_FUNCTION\_H

lz77\_encoding.c

#include <time.h>

#include "lz77\_encoding\_function.h"

int main(int argc, char \*\*argv) {

float offset\_avg, offset\_std, length\_avg, length\_std;

double compression\_time;

if(argc != 3) {

printf("You must supply 2 arguments: pgm image name, searching buffer size\n");

return 0;

}

char \*PGM\_image = argv[1];

unsigned int searching\_buffer\_size = atoi(argv[2]);

clock\_t begin = clock();

Encode\_Using\_LZ77(PGM\_image, searching\_buffer\_size, &offset\_avg, &offset\_std, &length\_avg, &length\_std);

clock\_t end = clock();

compression\_time = (double)(end - begin) / CLOCKS\_PER\_SEC;

printf("Offset average: %.2f\nOffset standard deviation: %.2f\nMatch length average: %.2f\nMatch length standard deviation: %.2f\nCompression time: %.2f\n", offset\_avg, offset\_std, length\_avg, length\_std, compression\_time);

}

lz\_decoding.c

#include <time.h>

#include "lz77\_decoding\_function.h"

int main(int argc, char \*\*argv) {

double compression\_time;

if(argc != 2) {

printf("You must supply 1 argument: an lz compressed file name\n");

return 0;

}

char \*lz\_image = argv[1];

clock\_t begin = clock();

Decode\_Using\_LZ77(lz\_image);

clock\_t end = clock();

compression\_time = (double)(end - begin) / CLOCKS\_PER\_SEC;

printf("Decompression time: %.2f\n", compression\_time);

}

mean\_absolute\_error.c

#include "mean\_absolute\_error.h"

float mean\_absolute\_error(char \*file\_name\_1\_ptr, char \*file\_name\_2\_ptr) {

int row, col, sum = 0;

float scale, abs\_error;

struct PGM\_Image pgmImage1, pgmImage2, \*pgmImage;

// Open the files

load\_PGM\_Image(&pgmImage1, file\_name\_1\_ptr);

load\_PGM\_Image(&pgmImage2, file\_name\_2\_ptr);

// Check that images are the same size

if(pgmImage1.height != pgmImage2.height || pgmImage1.width != pgmImage2.width) {

printf("Images do not have the same dimensions\n");

return 0;

}

// Check that images use the same gray value

if(pgmImage1.maxGrayValue != pgmImage2.maxGrayValue) {

// If pgm Image 2 has smaller gray value, divide for scale and multiply in loop

if(pgmImage1.maxGrayValue > pgmImage2.maxGrayValue) {

pgmImage = &pgmImage2;

scale = (float) pgmImage1.maxGrayValue / pgmImage2.maxGrayValue;

pgmImage2.maxGrayValue = pgmImage1.maxGrayValue;

} else {

pgmImage = &pgmImage1;

scale = (float) pgmImage2.maxGrayValue / pgmImage1.maxGrayValue;

pgmImage1.maxGrayValue = pgmImage2.maxGrayValue;

}

for(row = 0; row < pgmImage->height; row++)

for(col = 0; col < pgmImage->width; col++)

pgmImage->image[row][col] \*= scale;

}

// Calculate the absolute sum of differences between the images

for(row = 0; row < pgmImage1.height; row++)

for(col = 0; col < pgmImage1.width; col++)

sum += abs(pgmImage2.image[row][col] - pgmImage1.image[row][col]);

abs\_error = (float) sum / (row \* col);

return abs\_error;

}

mean\_absolute\_error.h

#ifndef MEAN\_ABSOLUTE\_ERROR\_H

#define MEAN\_ABSOLUTE\_ERROR\_H

#include <stdio.h>

#include <math.h>

#include "libpnm.h"

float mean\_absolute\_error(char \*file\_name\_1\_ptr, char \*file\_name\_2\_ptr);

#endif // MEAN\_ABSOLUTE\_ERROR\_H

compare\_pgm\_images.c

#include "mean\_absolute\_error.h"

int main(int argc, char \*\*argv) {

if(argc != 3) {

printf("You must supply 2 arguments: PGM file name 1, PGM file name 2\n");

return 0;

}

char \*pgmImage1 = argv[1],

\*pgmImage2 = argv[2];

float mae = mean\_absolute\_error(pgmImage1, pgmImage2);

printf("Mean absolute error: %.2f\n", mae);

}