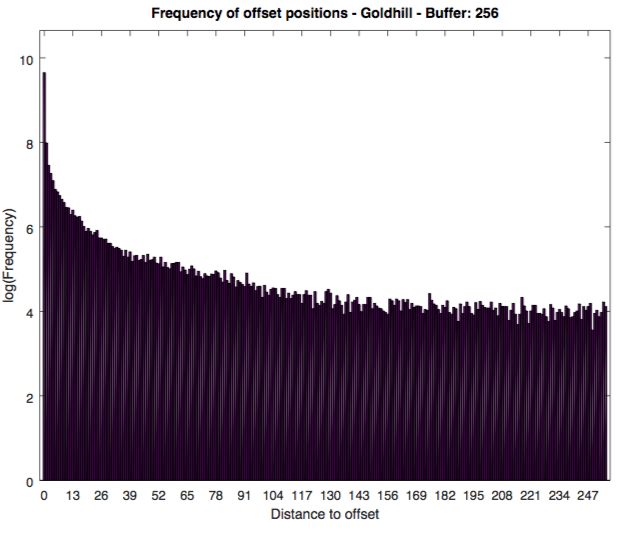
**\*\*\* README \*\*\***

I run an Ubuntu computer and could not use the makefile provided due to the linker for the math library being placed before the files it needs to link.

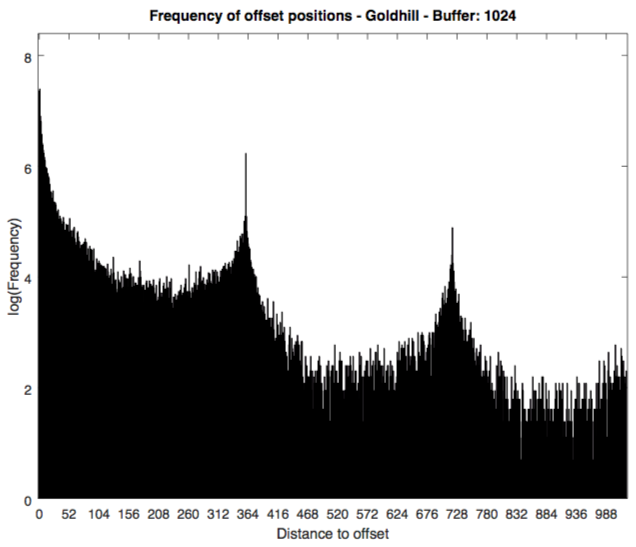
I am aware that this only causes a warning on Mac systems.

To fix this issue I added “-lm” at the end of the following lines in the makefile: 30, 46

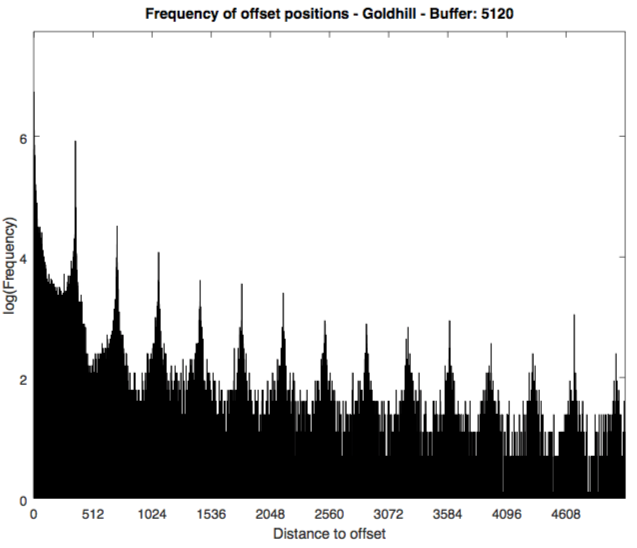
**\*\*\* ANALYSIS \*\*\***



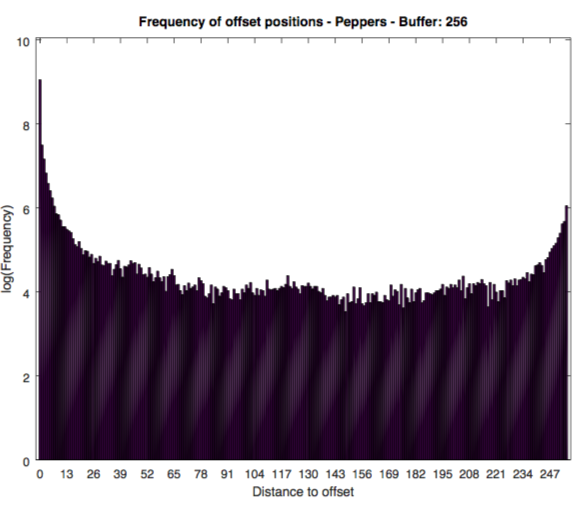
**Figure 1.** Histogram displaying the logarithmic frequency of the offset distance comprising the tokens generated for the Goldhill image with a buffer size of 256.

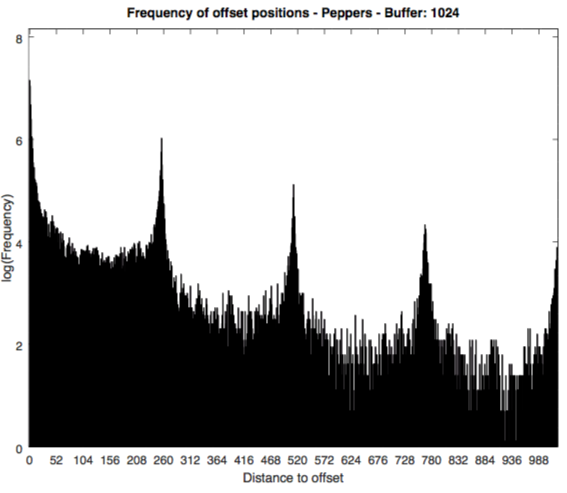


**Figure 2.** Histogram displaying the logarithmic frequency of the offset distance comprising the tokens generated for the Goldhill image with a buffer size of 1024.

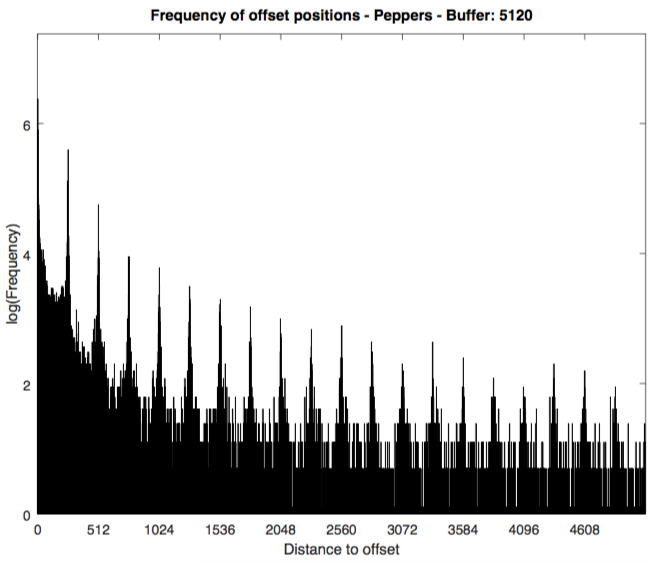


**Figure 3.** Histogram displaying the logarithmic frequency of the offset distance comprising the tokens generated for the Goldhill image with a buffer size of 5120.

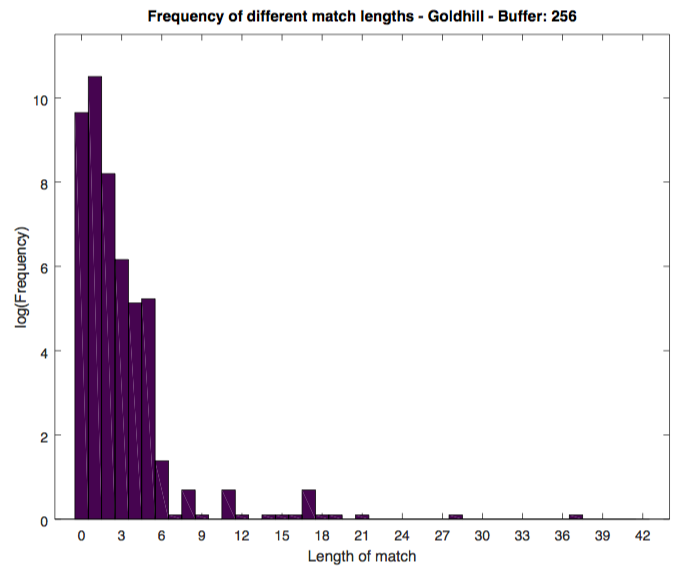
 **Figure 4.** Histogram displaying the logarithmic frequency of the offset distance comprising the tokens generated for the Peppers image with a buffer size of 256.



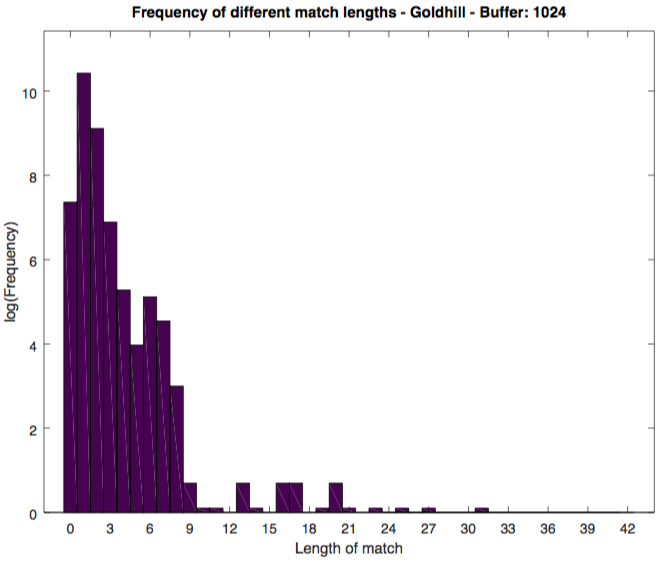
**Figure 5.** Histogram displaying the logarithmic frequency of the offset distance comprising the tokens generated for the Peppers image with a buffer size of 1024.



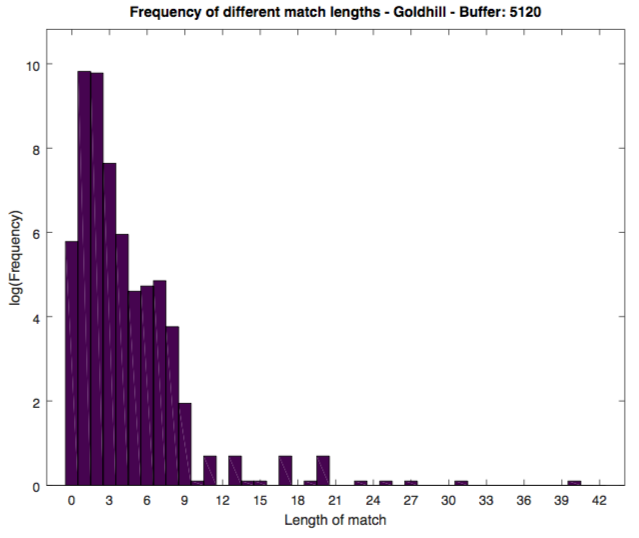
**Figure 6.** Histogram displaying the logarithmic frequency of the offset distance comprising the tokens generated for the Peppers image with a buffer size of 5120.



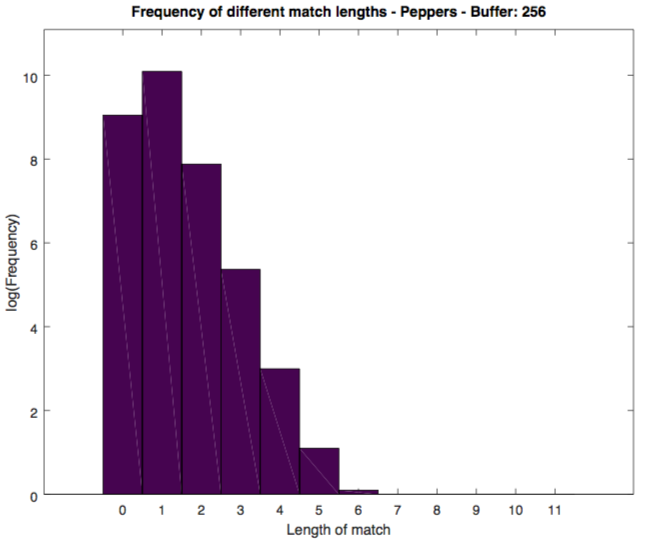
**Figure 7.** Histogram displaying the logarithmic frequency of the match lengths comprising the tokens generated for the Goldhill image with a buffer size of 256.



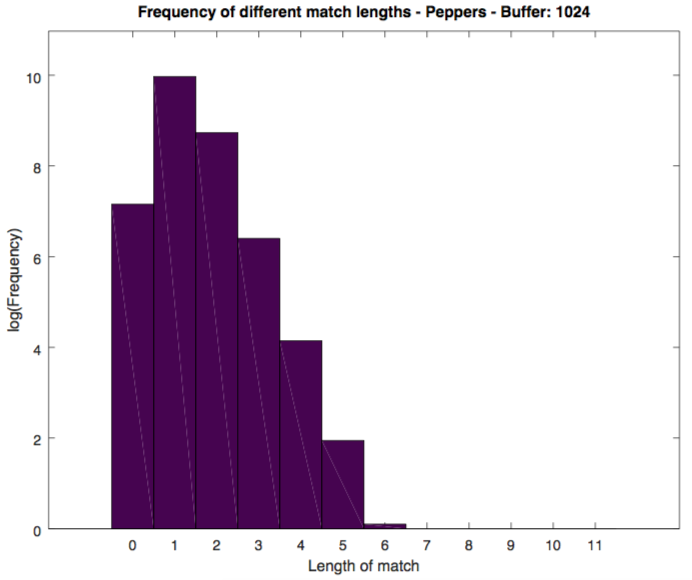
**Figure 8.** Histogram displaying the logarithmic frequency of the match lengths comprising the tokens generated for the Goldhill image with a buffer size of 1024.



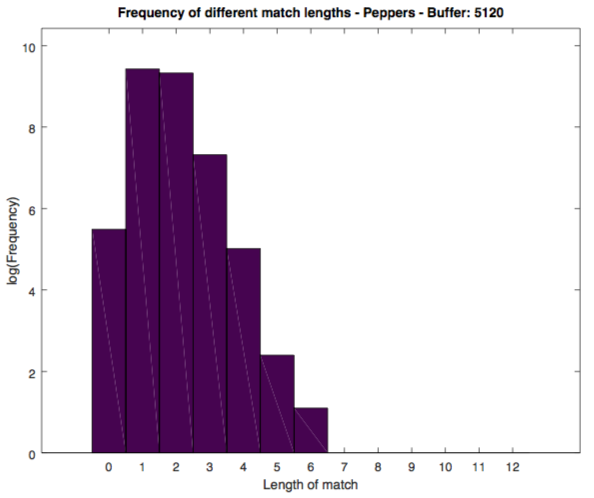
**Figure 9.** Histogram displaying the logarithmic frequency of the match lengths comprising the tokens generated for the Goldhill image with a buffer size of 5120.



**Figure 10.** Histogram displaying the logarithmic frequency of the match lengths comprising the tokens generated for the Peppers image with a buffer size of 256.



**Figure 11.** Histogram displaying the logarithmic frequency of the match lengths comprising the tokens generated for the Peppers image with a buffer size of 1024.



**Figure 12.** Histogram displaying the logarithmic frequency of the match lengths comprising the tokens generated for the Peppers image with a buffer size of 5120.

**Comments on the shape of the offsets histograms (Figures 1–6):**

In each of the offset histograms (except the first figure 1) we see peaks which occur periodically. The number of peaks that can be observed increases as the size of the buffer increases.

We can see that for each image, the distance between the peaks remains constant, such that the images with buffers of large sizes represent a continuation of the same trend seen in previous figures.

For instance, the distance between peaks for the goldhill image is approximately 360, which explains why figure 1 does not display such a peak: the buffer is too short to reach this point.

The peaks (or impulses) of this system represent the width of the image. LZ77 encodes from left to right, top to bottom, as if the image were linearized, however when it reaches the right side of the image it will loop back to the left. Since it is common for two adjacent pixels to be equal/similar, there is an increased likelihood that in the buffer if you look back in the buffer by the same number of positions as the size of the image width that you will find a match.

Of course, this is not always the case as you move down a picture, which is why we see a decrease in the intensity of the images as you move further backwards through the buffer.

**Comments on the shape of the lengths histograms (Figures 7–12):**

As the buffer gets larger we see a tendency to favour longer match lengths, as there is a larger pool from which a match can be found increasingly the likelihood of this.

As expected, lower match lengths are seen more prevalently than longer match lengths due to the probability of certain sequences appearing in the past.

The goldhill image contains much larger match lengths than the pepper image, suggesting that there are a number of repeated elements in the goldhill image, things such as the roadway or other bricks.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Image Type** | **Buffer Size** | **Encoding Time (ms)** | **Average** | | **Standard Deviation** | |
| **Offset** | **Length** | **Offset** | **Length** |
| Peppers | 256 | 122 | 73.32 | 0.85 | 0.0455 | 0.0049 |
| 1024 | 369 | 239.44 | 1.22 | 0.0900 | 0.0064 |
| 5120 | 1463 | 1038.05 | 1.57 | 0.2017 | 0.0078 |
| Goldhill | 256 | 191 | 45.36 | 0.84 | 0.0283 | 0.0038 |
| 1024 | 580 | 213.22 | 1.27 | 0.0683 | 0.0053 |
| 5120 | 2363 | 1044.65 | 1.64 | 0.1631 | 0.0065 |

**Figure 13.** Data collected by applying LZ77 encoding to two different images, one of peppers the other of a landscape. Reported values include the time it took to encode the images in milliseconds, along with the averages and standard deviations of both the offset (the value assigned to each LZ77 token indicated the offset from the frontier) and the length (the number of consecutive matching positions found starting at the offset).

**Comments on which buffer size is the most appropriate:**

As the aim of this algorithm is compression, I believe that choosing the buffer size that will give the best rate of compression is the most suitable.

Therefore, for each image the 5120 buffer is the most suitable. For each image, the average length of the offset is greater in the 5120 buffer trials, even when accounting for the increased standard deviation associated with these trials. Since within one standard deviation the length of the matches is still greater than that of the alternates, this seems to be the clear choice.

A final consideration is time, and although the 5120 buffer takes significantly longer than each of the alternates, I believe that this is an acceptable tradeoff to achieve better compression.

**\*\*\* CODE \*\*\***

**lz77\_encoding\_function.h**

#include <stdlib.h>

#include <stdio.h>

#include <unistd.h>

#include <string.h>

#include <math.h>

#include "libpnm.h"

#ifndef \_ENCODE\_USING\_LZ77\_\_

#define \_ENCODE\_USING\_LZ77\_\_

struct header{

unsigned int max\_offset;

int number\_of\_tokens;

int width;

int height;

int max\_gray\_value;

};

int min(int a, int b);

unsigned char get\_pgm\_image\_value(struct PGM\_Image\* image, int pos);

int find\_number\_of\_matches(struct PGM\_Image\* image, int match\_pos, int text\_pos, int total);

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.c**

#include "lz77\_encoding\_function.h"

// Return the max of two integers

int min(int a, int b){

return a > b ? b : a;

}

// Using a single integer position, determine the character referenced in the pgm image

unsigned char get\_pgm\_image\_value(struct PGM\_Image\* image, int pos){

return image->image[(int)floor(pos / image->width)][pos % image->width];

}

// Determine the number of matching characters when using an offset of match\_pos

// and a frontier of text\_pos

int find\_number\_of\_matches(struct PGM\_Image\* image, int match\_pos, int text\_pos, int total){

int match\_num = 0;

while(get\_pgm\_image\_value(image, match\_pos) == get\_pgm\_image\_value(image, text\_pos)){

match\_num++;

match\_pos++;

text\_pos++;

if(match\_pos >= total || text\_pos >= total){

break;

}

}

return match\_num;

}

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){

// loading the image

struct PGM\_Image\* image = malloc(sizeof(struct PGM\_Image));

load\_PGM\_Image(image, in\_PGM\_filename\_Ptr);

int total = image->width \* image->height;

// Storing the token data

int\* offsets = calloc(total, sizeof(int));

int\* match\_lengths = calloc(total, sizeof(int));

int\* mismatches = malloc(total \* sizeof(int));

int number\_of\_tokens = 0;

int max\_match\_pos; // offset

int max\_match\_num; // match length

int cur\_pos = 0; // current position within the image

int cur\_value;

/\*\*\* DETERMINING TOKENS \*\*\*/

while(cur\_pos < total - 1){

max\_match\_num = 0;

max\_match\_pos = 0;

// Get the next value in the image

cur\_value = get\_pgm\_image\_value(image, cur\_pos);

// Searching backwards through the buffer

for(int i = 1; i <= min(cur\_pos, searching\_buffer\_size); i++){

// If we find a match

if(get\_pgm\_image\_value(image, cur\_pos - i) == cur\_value){

// If we find the match matches a larger number than previously seen update max values

int temp = find\_number\_of\_matches(image, cur\_pos - i, cur\_pos, total);

if(temp > max\_match\_num){

max\_match\_num = temp;

max\_match\_pos = i;

}

}

}

// Update the token information

offsets[number\_of\_tokens] = max\_match\_pos; // offset away

match\_lengths[number\_of\_tokens] = max\_match\_num; // how long the match was

if(cur\_pos + max\_match\_num == total){

mismatches[number\_of\_tokens++] = EOF;

} else{

mismatches[number\_of\_tokens++] = get\_pgm\_image\_value(image, cur\_pos + max\_match\_num); // the value after the match

}

// Update the current position to be the position after the next mismatch

cur\_pos += max\_match\_num + 1;

}

/\*\*\* SAVING COMPRESSED IMAGE + HEADER \*\*\*/

// Building the header

struct header\* file\_header = malloc(sizeof(struct header));

file\_header->max\_offset = searching\_buffer\_size;

file\_header->number\_of\_tokens = number\_of\_tokens;

file\_header->width = image->width;

file\_header->height = image->height;

file\_header->max\_gray\_value = MAX\_GRAY\_VALUE;

// Creating altered filename for output (lz)

char numstr1[100];

sprintf(numstr1, "%s.%d.lz", in\_PGM\_filename\_Ptr, searching\_buffer\_size);

FILE \*f = fopen(numstr1, "wb");

// Writing to file

fwrite(file\_header, sizeof(struct header), 1, f);

fwrite(offsets, sizeof(int), number\_of\_tokens, f);

fwrite(match\_lengths, sizeof(int), number\_of\_tokens, f);

fwrite(mismatches, sizeof(int), number\_of\_tokens, f);

fclose(f);

/\*\*\* OFFSETS \*\*\*/

// Determining the frequencies of all of the offsets

int\* offset\_frequencies = calloc(searching\_buffer\_size, sizeof(int));

for(int i = 0; i < number\_of\_tokens; i++){

offset\_frequencies[offsets[i]]++;

}

// Creating altered filename for output (csv)

char numstr2[100];

sprintf(numstr2, "%s.%d.offsets.csv", in\_PGM\_filename\_Ptr, searching\_buffer\_size);

// Writing the offsets

f = fopen(numstr2, "w");

for(int i = 0; i < searching\_buffer\_size; i++){

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

fprintf(f, "%d, %d\n", i, offset\_frequencies[i]);

}

}

fclose(f);

/\*\*\* MATCH LENGTHS \*\*\*/

// Determining the range of match lengths

int max\_match\_length = 0;

for(int i = 0; i < number\_of\_tokens; i++){

if(match\_lengths[i] > max\_match\_length){

max\_match\_length = match\_lengths[i];

}

}

// Determining the frequencies of match lengths

int\* match\_length\_frequencies = calloc(max\_match\_length + 1, sizeof(int));

for(int i = 0; i < number\_of\_tokens; i++){

match\_length\_frequencies[match\_lengths[i]]++;

}

// // Creating altered filename for output (csv)

char numstr3[100];

sprintf(numstr3, "%s.%d.lengths.csv", in\_PGM\_filename\_Ptr, searching\_buffer\_size);

// Writing the match lengths

f = fopen(numstr3, "w");

for(int i = 0; i < max\_match\_length; i++){

if(match\_length\_frequencies[i] != 0){

fprintf(f, "%d, %d\n", i, match\_length\_frequencies[i]);

}

}

fclose(f);

/\*\*\* AVERAGE + STANDARD DEVIATION \*\*\*/

// Calculating the average

\*avg\_offset\_Ptr = 0.0;

\*avg\_length\_Ptr = 0.0;

for(int i = 0; i < number\_of\_tokens; i++){

\*avg\_offset\_Ptr += offsets[i];

\*avg\_length\_Ptr += match\_lengths[i];

}

\*avg\_offset\_Ptr /= number\_of\_tokens;

\*avg\_length\_Ptr /= number\_of\_tokens;

// Calculating standard deviation

\*std\_offset\_Ptr = 0.0;

\*std\_length\_Ptr = 0.0;

for(int i = 0; i < number\_of\_tokens; i++){

\*std\_offset\_Ptr += pow(offsets[i] - \*avg\_offset\_Ptr, 2);

\*std\_length\_Ptr += pow(match\_lengths[i] - \*avg\_length\_Ptr, 2);

}

\*std\_offset\_Ptr = sqrt(\*avg\_offset\_Ptr / number\_of\_tokens);

\*std\_length\_Ptr = sqrt(\*avg\_length\_Ptr / number\_of\_tokens);

// Clearing all allocated memory

free(offsets);

free(match\_lengths);

free(mismatches);

free(file\_header);

free(image);

free(match\_length\_frequencies);

free(offset\_frequencies);

}

**lz77\_decoding\_function.h**

#include <stdlib.h>

#include <stdio.h>

#include <unistd.h>

#include <string.h>

#include <math.h>

#include "libpnm.h"

#include "lz77\_encoding\_function.h"

#ifndef \_DECODE\_USING\_LZ77\_\_

#define \_DECODE\_USING\_LZ77\_\_

void save\_pgm\_image\_value(struct PGM\_Image\* image, int pos, unsigned char value);

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

#endif

**lz77\_decoding\_function.c**

#include "lz77\_decoding\_function.h"

#include "lz77\_encoding\_function.c"

// Save a value based on 1D number in 2D PGM image

void save\_pgm\_image\_value(struct PGM\_Image\* image, int pos, unsigned char value){

image->image[(int)floor(pos / image->width)][pos % image->width] = value;

}

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

/\*\*\* READING THE COMPRESSED FILE \*\*\*/

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

// Reading the header

struct header\* file\_header = malloc(sizeof(struct header));

fread(file\_header, sizeof(struct header), 1, f);

// Retrieving relevant values from the header

int number\_of\_tokens = file\_header->number\_of\_tokens;

// Creating a PGM image to populate

struct PGM\_Image\* image = malloc(sizeof(struct PGM\_Image));

create\_PGM\_Image(image, file\_header->width, file\_header->height, file\_header->max\_gray\_value);

free(file\_header);

// Retrieving the tokens

int\* offsets = malloc(sizeof(int) \* number\_of\_tokens);

int\* match\_lengths = malloc(sizeof(int) \* number\_of\_tokens);

int\* mismatches = malloc(sizeof(int) \* number\_of\_tokens);

fread(offsets, sizeof(int), number\_of\_tokens, f);

fread(match\_lengths, sizeof(int), number\_of\_tokens, f);

fread(mismatches, sizeof(int), number\_of\_tokens, f);

fclose(f);

/\*\*\* DECOMPRESSING THE IMAGE \*\*\*/

int cur\_pos = 0;

for(int i = 0; i < number\_of\_tokens; i++){

for(int j = offsets[i]; j > offsets[i] - match\_lengths[i]; j--){

// Adjusting for non positive indices

int pos = cur\_pos - j;

if(j <= 0){

pos = cur\_pos - offsets[i] + (j % offsets[i]);

}

save\_pgm\_image\_value(image, cur\_pos, get\_pgm\_image\_value(image, pos));

cur\_pos++;

}

// Adding the mismatch

if(mismatches[i] != -1){

save\_pgm\_image\_value(image, cur\_pos, mismatches[i]);

}

cur\_pos++;

}

/\*\*\* SAVING THE IMAGE \*\*\*/

char new\_file\_name[100];

sprintf(new\_file\_name, "%s.pgm", in\_compressed\_filename\_Ptr);

save\_PGM\_Image(image, new\_file\_name, 1);

free(offsets);

free(match\_lengths);

free(mismatches);

free(image);

}

**lz77\_encoding.c**

#include <stdlib.h>

#include <stdio.h>

#include <unistd.h>

#include <string.h>

#include <time.h>

#include "lz77\_encoding\_function.h"

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

if(argc != 3){

printf("Please ensure you have entered two arguments. You entered: %d\n", argc - 1);

exit(0);

}

if(access(argv[1], F\_OK) == -1){

printf("Please ensure you have entered a valid file.\n");

exit(0);

}

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

if(argv[2][0] != '0' && searching\_buffer\_size == 0){

printf("Please ensure you have entered a buffer size.\n");

exit(0);

}

float avg\_offset\_Ptr, std\_offset\_Ptr, avg\_length\_Ptr, std\_length\_Ptr;

clock\_t time = clock();

Encode\_Using\_LZ77(argv[1], searching\_buffer\_size, &avg\_offset\_Ptr, &std\_offset\_Ptr,

&avg\_length\_Ptr, &std\_length\_Ptr);

time = clock() - time;

printf("\nCompression time: %ld ms\n", (time \* 1000) / CLOCKS\_PER\_SEC);

printf("Average offset: %f\n", avg\_offset\_Ptr);

printf("Stand deviation of offset: %f\n", std\_offset\_Ptr);

printf("Average length of match: %f\n", avg\_length\_Ptr);

printf("Stand deviation of match length: %f\n\n", std\_length\_Ptr);

return 0;

}

**lz77\_decoding.c**

#include <stdlib.h>

#include <stdio.h>

#include <unistd.h>

#include <string.h>

#include <time.h>

#include "lz77\_decoding\_function.h"

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

if(argc != 2){

printf("Please ensure you have entered two arguments. You entered: %d\n", argc - 1);

exit(0);

}

if(access(argv[1], F\_OK) == -1){

printf("Please ensure you have entered a valid file.\n");

exit(0);

}

clock\_t time = clock();

Decode\_Using\_LZ77(argv[1]);

time = clock() - time;

printf("\nDecompression Time: %ld ms\n\n", (time \* 1000) / CLOCKS\_PER\_SEC);

return 0;

}

**mean\_absolute\_error.h**

#include <stdlib.h>

#include <stdio.h>

#include <unistd.h>

#include <string.h>

#include "libpnm.h"

#ifndef \_MEAN\_ABSOLUTE\_ERROR\_

#define \_MEAN\_ABSOLUTE\_ERROR\_

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

#endif

**mean\_absolute\_error.c**

#include "mean\_absolute\_error.h"

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

struct PGM\_Image\* image1= malloc(sizeof(struct PGM\_Image));

if(load\_PGM\_Image(image1, file\_name\_1\_ptr) == -1){

printf("Please ensure you have entered a valid input file.\n");

exit(0);

}

struct PGM\_Image\* image2 = malloc(sizeof(struct PGM\_Image));

if(load\_PGM\_Image(image2, file\_name\_2\_ptr) == -1){

printf("Please ensure you have entered a valid input file.\n");

exit(0);

}

if(image1->width != image2->width || image1->height != image2->height){

printf("Images must be of the same dimensions.\n");

exit(0);

}

float factor1 = 1;

float factor2 = 1;

if(image1->maxGrayValue < image2->maxGrayValue){

factor1 = image2->maxGrayValue / image1->maxGrayValue;

} else if (image1->maxGrayValue > image2->maxGrayValue){

factor2 = image1->maxGrayValue / image2->maxGrayValue;

}

float error = 0;

float temp;

for(int h = 0; h < image1->height; h++){

for(int w = 0; w < image1->width; w++){

temp = (image1->image[h][w] \* factor1) - (image2->image[h][w] \* factor2);

if(temp < 0){

temp \*= -1;

}

error += temp;

}

}

return error / (image1->width \* image1->height);

}

**compare\_pgm\_images.c**

#include <stdlib.h>

#include <stdio.h>

#include <unistd.h>

#include <string.h>

#include "libpnm.h"

#include "mean\_absolute\_error.h"

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

// Checking args

if(argc != 3){

printf("Please ensure you have entered two arguments. You entered: %d\n", argc - 1);

exit(0);

}

// Checking that args represent valid files

if(access(argv[1], F\_OK) == -1 || access(argv[1], F\_OK) == -1){

printf("Please ensure you have entered two valid files.\n");

exit(0);

}

// Checking the mean absolute error

float error = mean\_absolute\_error(argv[1], argv[2]);

printf("\nThe mean absolute error between the two images is: %f\n\n", error);

return 0;

}