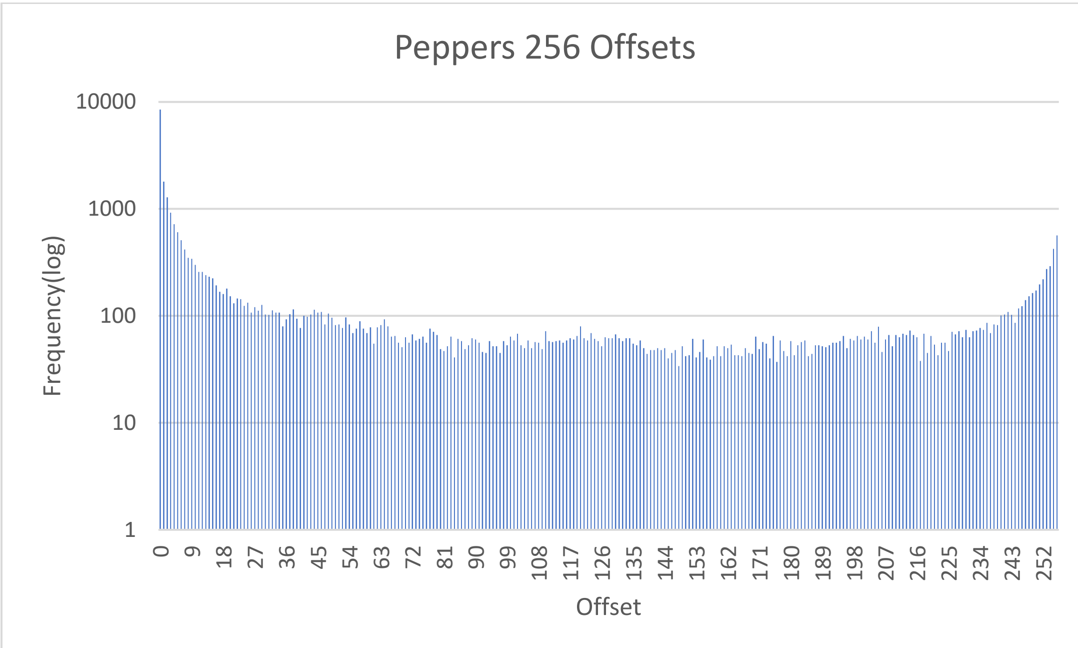
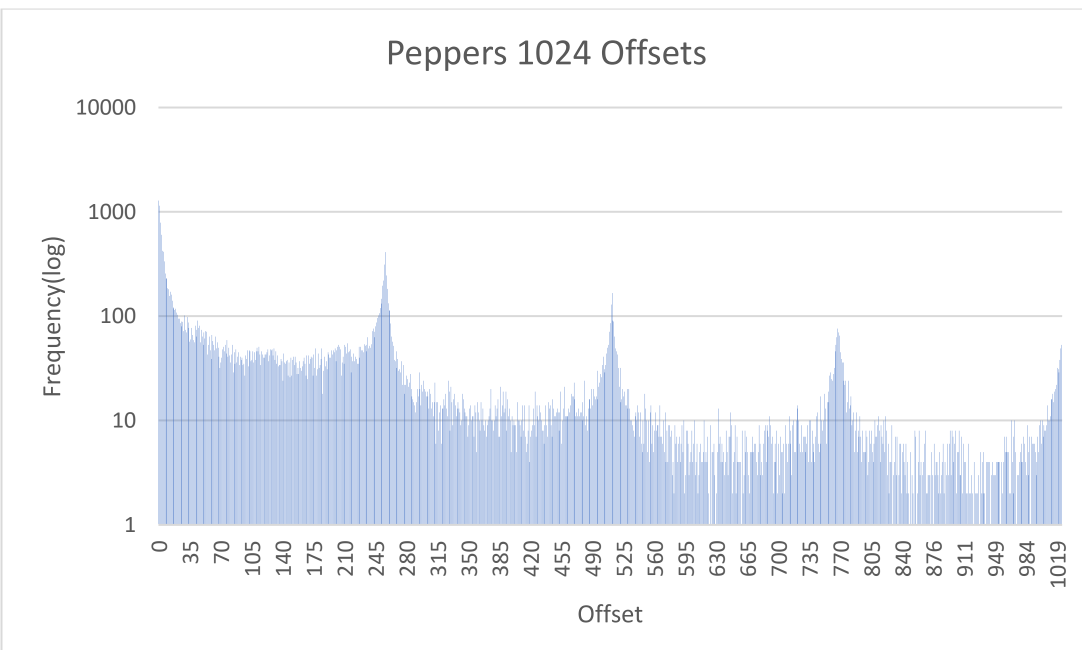
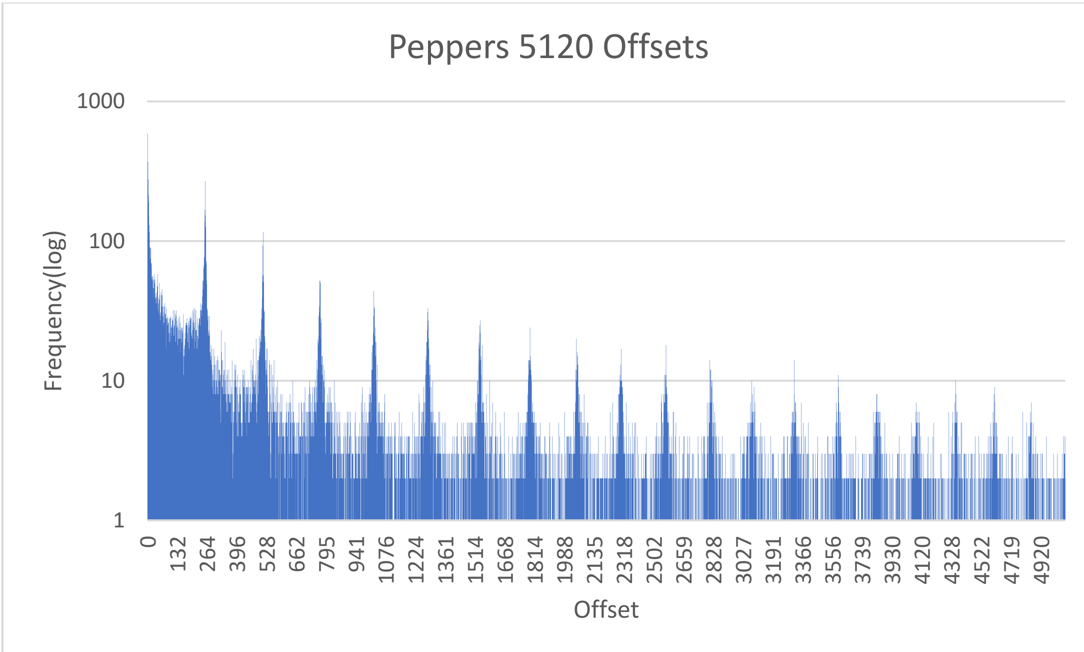
CS 4481b Assignment 3 Report

Paul Bartlett

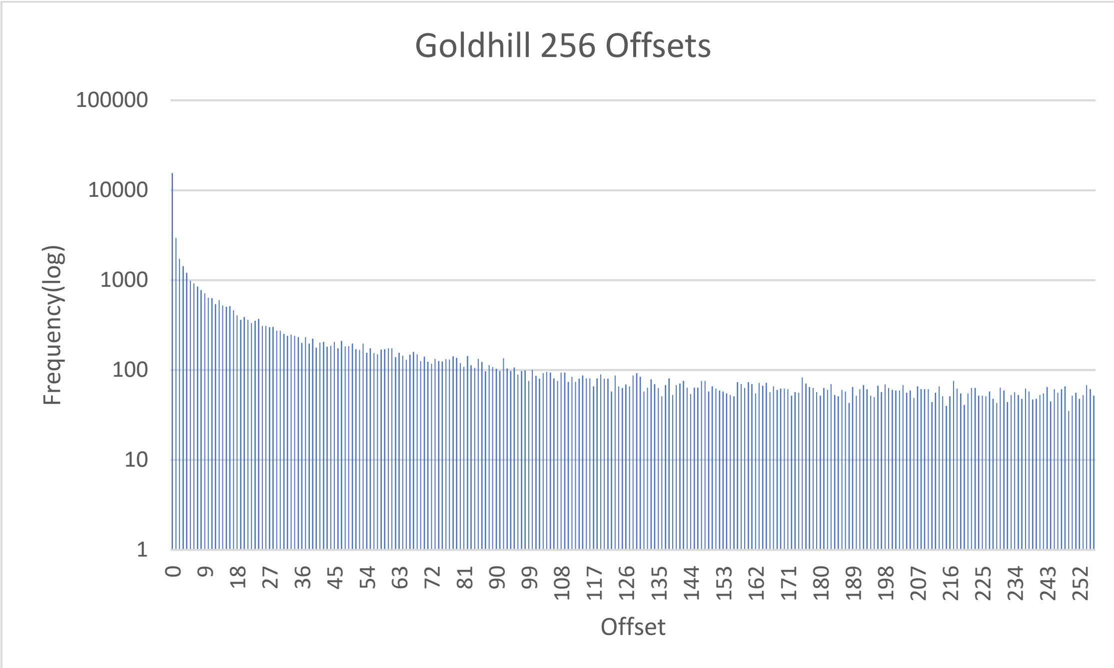
Peppers Offset Histograms

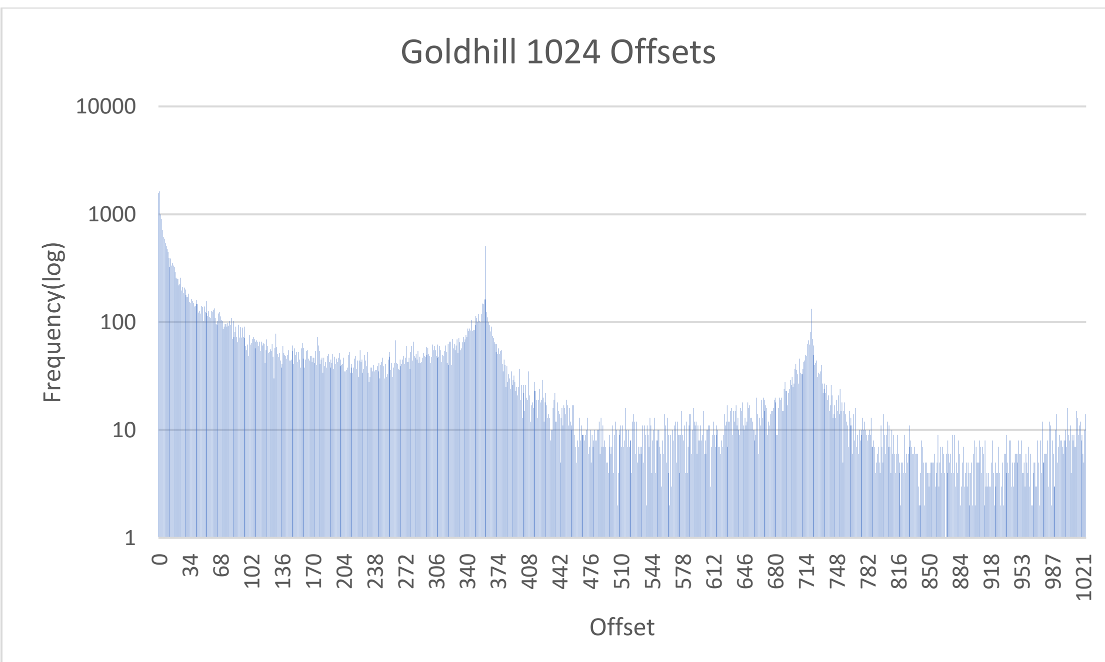


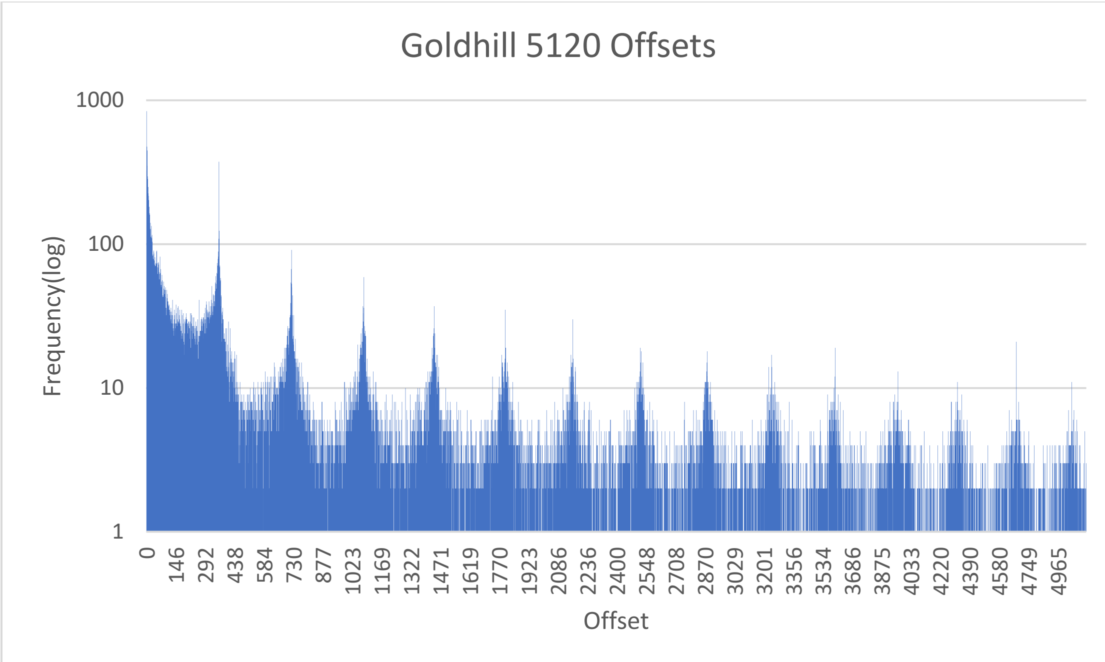



Peppers 5120 Offsets
1000
100
10
1
Frequency(log)
0
132
264
396
528
662
795
941
1076
1224
1361
1514
1668
1814
1988
2135
2318
2502
2659
2828
3027
3191
3366
3556
3739
3930
4120
4328
4522
4719
4920
Offset

Goldhill Offset Histograms

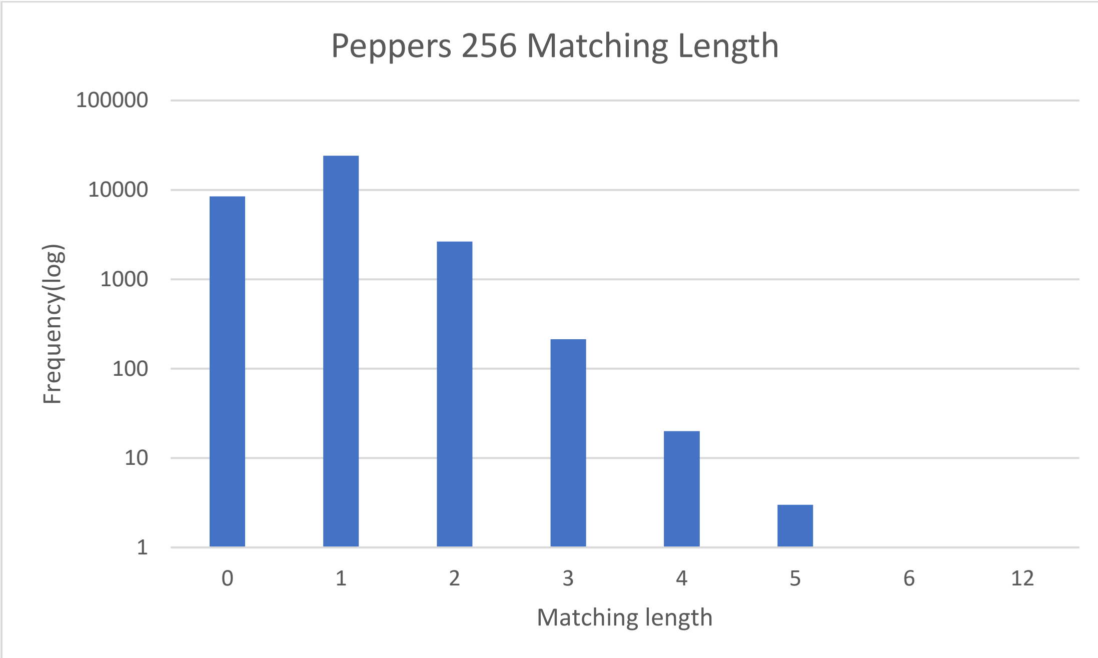


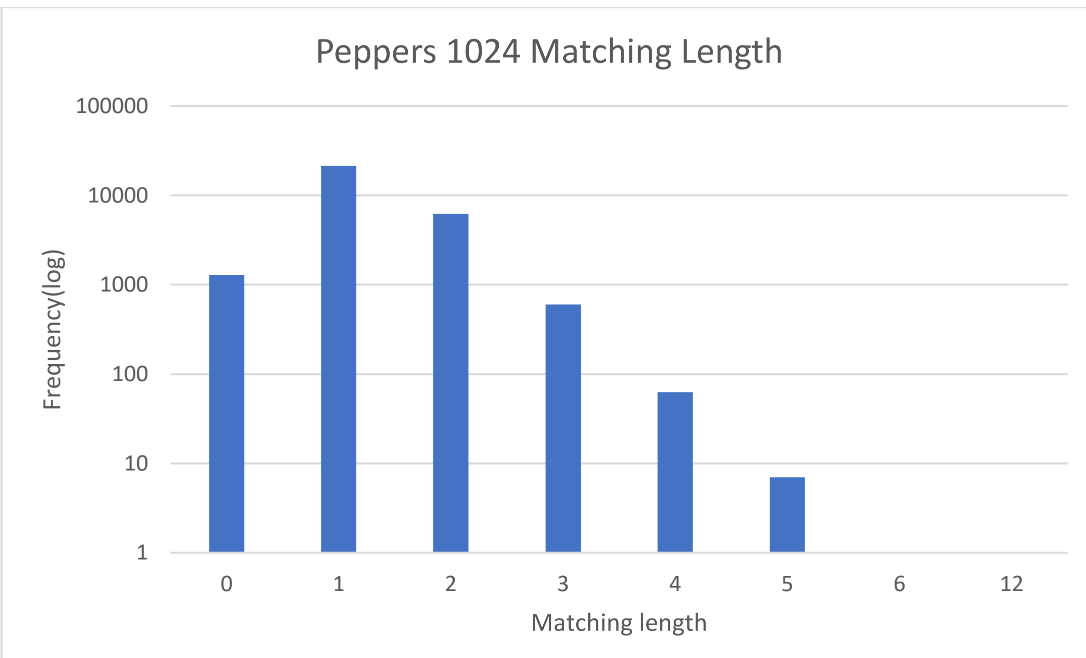


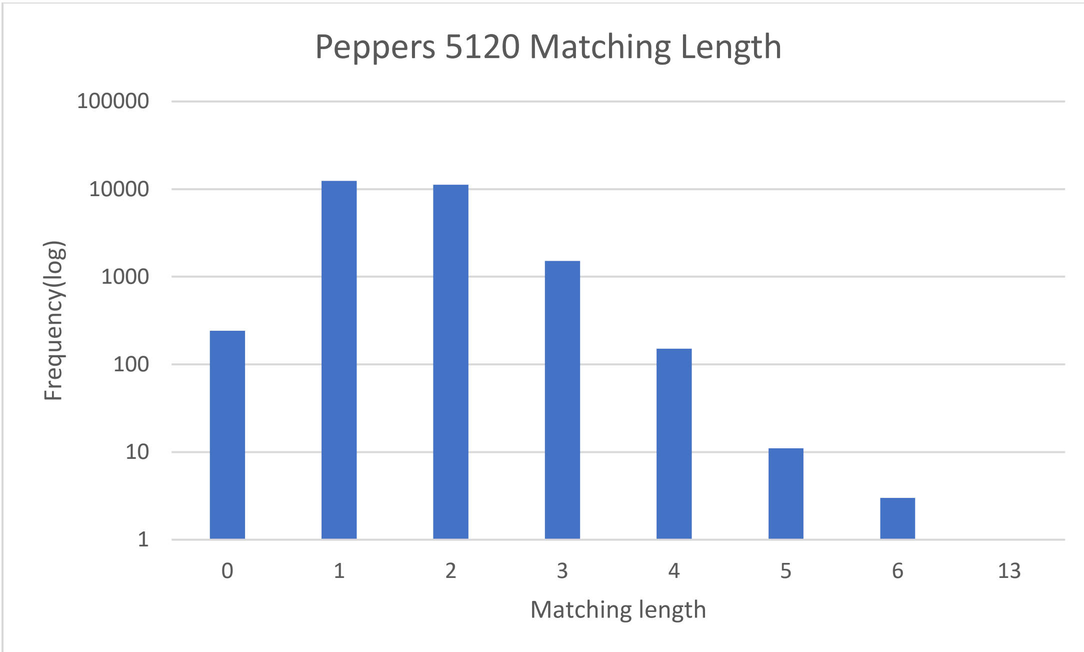


Comments: The images with larger offsets show the same impulse pattern for both images. This is because the width of the Goldhill and Peppers images are 360 and 256 respectively, and there is a greater chance of matching a pixel in the same proximity as the original pixel. This causes the frequency of 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

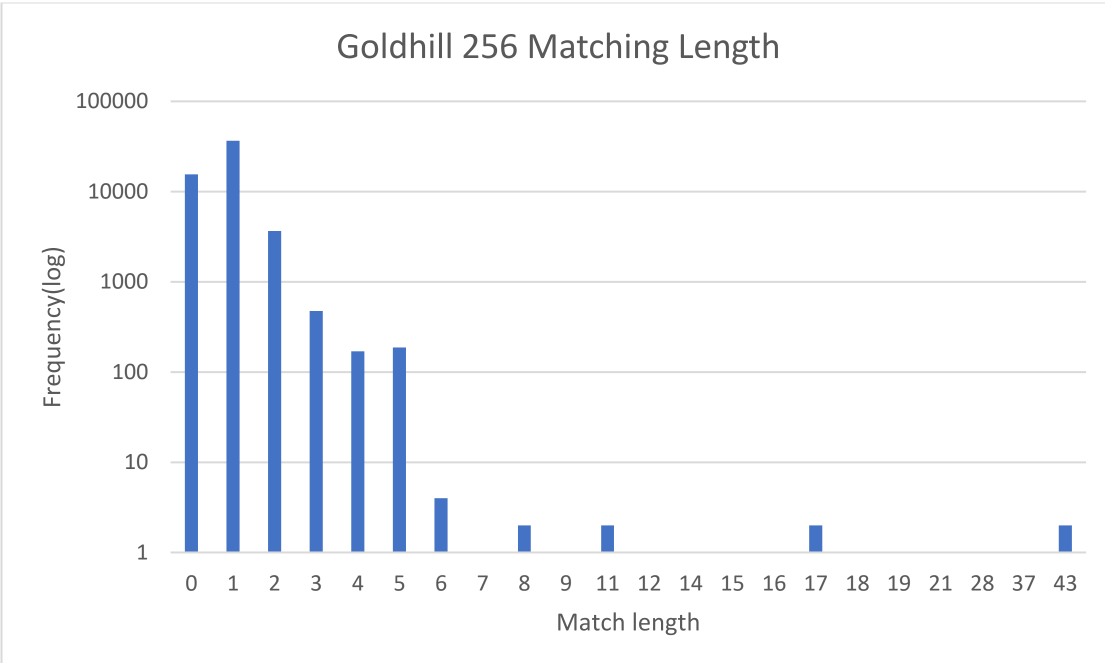


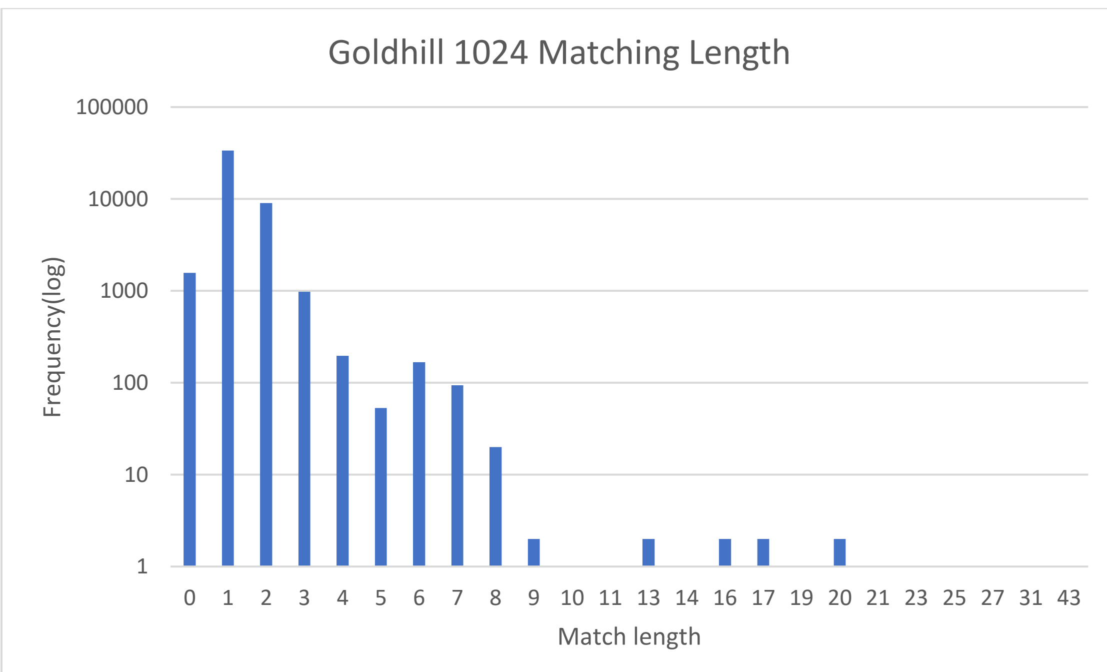


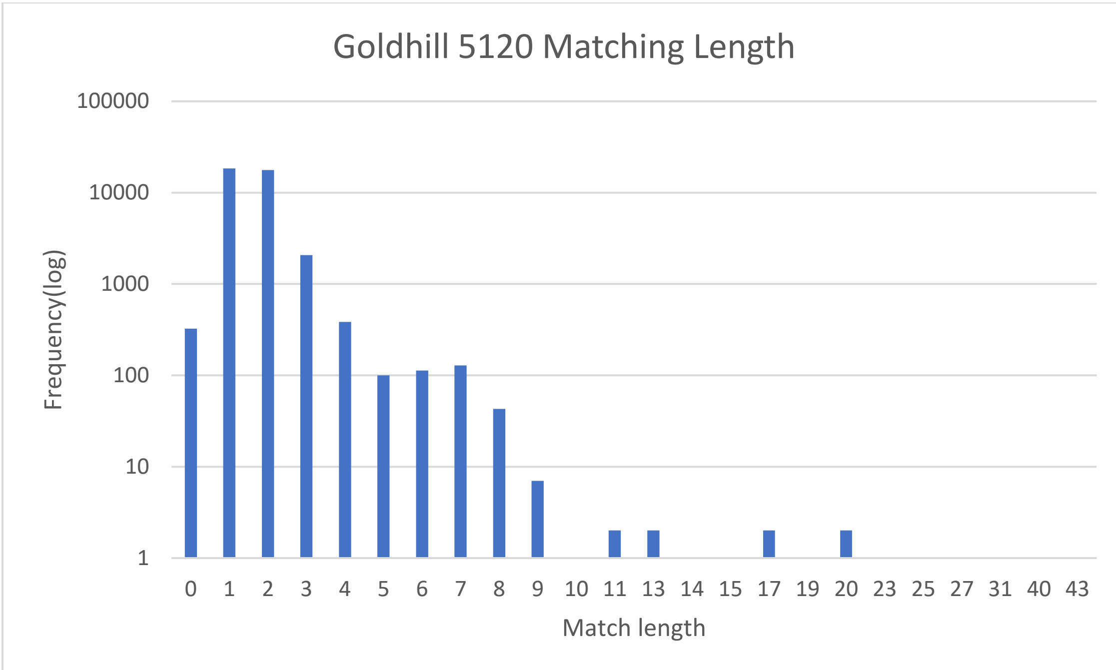


Peppers 5120 Matching Length
100000
10000
1000
100
10
1
Frequency(log)
0
1
2
3
4
5
6
13
Matching length

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

```c
#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

```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

```c
#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);
}
```