SFWRENG 2S03 Assignment 4

Caleb Mech and Matthew Williams

30 November 2018

Code

readwriteppm.c

```
#include <stdio.h>
#include <stdlib.h>
#include "./a4.h"

#define data(row, col) image->data[row * image->width + col]

PPM_IMAGE *read_ppm(const char *file_name) {
  FILE *file = fopen(file_name, "r");
  PPM_IMAGE *image = malloc(sizeof(PPM_IMAGE));

  fscanf(file, "%*s\n");  // Ignore Netpbm format string
  fscanf(file, "%d␣%d\n", &image->width, &image->height);
  fscanf(file, "%d\n", &image->max_color);

  image->data = malloc(sizeof(PIXEL) * image->width * image->height);

  for (int row = 0; row < image->height; row++) {
    for (int col = 0; col < image->width; col++) {
      fscanf(file, "%hhd␣%hhd␣%hhd␣", &data(row, col).r, &data(row, col).g,
             &data(row, col).b);
    }
    fscanf(file, "\n");
  }

  fclose(file);
  return image;
}

void write_ppm(const char *file_name, const PPM_IMAGE *image) {
  FILE *file = fopen(file_name, "w");

  fprintf(file, "%s\n", "P3");
  fprintf(file, "%d␣%d\n", image->width, image->height);
  fprintf(file, "%d\n", image->max_color);

  for (int row = 0; row < image->height; row++) {
    for (int col = 0; col < image->width; col++) {
      fprintf(file, "%d␣%d␣%d␣", data(row, col).r, data(row, col).g,
              data(row, col).b);
    }
    fprintf(file, "\n");
  }
```

```
  fclose(file);
}
```

fitness.c

```
#include <math.h>
#include "./a4.h"

double comp_distance(const PIXEL *A, const PIXEL *B, int image_size) {
  double d = 0.0;

  for (int i = 0; i < image_size; i++) {
    d += pow(A[i].r - B[i].r, 2)
       + pow(A[i].g - B[i].g, 2)
       + pow(A[i].b - B[i].b, 2);
  }

  return sqrt(d);
}

void comp_fitness_population(const PIXEL *image, Individual *individual,
                             int population_size) {
  // Assuming all images are the same size
  int image_size = individual[0].image.width * individual[0].image.height;
  for (int i = 0; i < population_size; i++) {
    individual[i].fitness =
        comp_distance(image, individual[i].image.data, image_size);
  }
}
```

population.c

```
#include <stdlib.h>
#include "./a4.h"

#define data(row, col) data[row * width + col]

PIXEL *generate_random_image(int width, int height, int max_color) {
  PIXEL *data = malloc(sizeof(PIXEL) * width * height);

  for (int row = 0; row < height; row++) {
    for (int col = 0; col < width; col++) {
      data(row, col).r = rand() % (max_color + 1);
      data(row, col).g = rand() % (max_color + 1);
      data(row, col).b = rand() % (max_color + 1);
    }
  }

  return data;
}

Individual *generate_population(int population_size, int width, int height,
                                int max_color) {
  Individual *population = malloc(sizeof(Individual) * population_size);

  for (int i = 0; i < population_size; i++) {
    population[i].image.width = width;
    population[i].image.height = height;
```

```
    population[i].image.max_color = max_color;
    population[i].image.data = generate_random_image(width, height, max_color);
    population[i].fitness = 0.0;
  }

  return population;
}
```

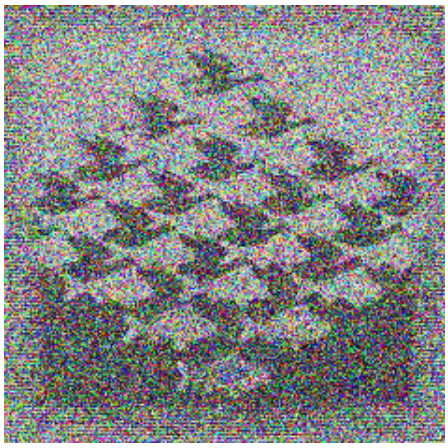
evolve.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "./a4.h"

// Sorts in ascending order
int comparator(const void *p, const void *q) {
  double l = ((Individual *)p)->fitness;
  double r = ((Individual *)q)->fitness;
  return (l - r);
}

PPM_IMAGE *evolve_image(const PPM_IMAGE *image, int num_generations,
                        int population_size, double rate) {
  // Take in parameters
  int height = image->height, width = image->width,
      max_color = image->max_color;
  // Generate Population
  Individual *population =
      generate_population(population_size, width, height, max_color);
  // Compute Fitness
  comp_fitness_population(image->data, population, population_size);
  // Sort population
  qsort(population, population_size, sizeof(*population), comparator);
  for (int i = 1; i < num_generations; i++) {
    // crossover, mutate then resort
    crossover(population, population_size);
    mutate_population(population, population_size, rate);
    comp_fitness_population(image->data, population, population_size);
    qsort(population, population_size, sizeof(*population), comparator);
  }
  // Free data from images
  for (int i = 1; i < population_size; i++)
    free(population[i].image.data);

  // Make copy of best image before destroying population
  PPM_IMAGE *final_image = malloc(sizeof(PPM_IMAGE));
  memcpy(final_image, &(population[0].image), sizeof(PPM_IMAGE));
  final_image->data = population[0].image.data;

  free(population);

  return final_image;
}

void free_image(PPM_IMAGE *p) { free(p->data), free(p); }
```

mutate.c

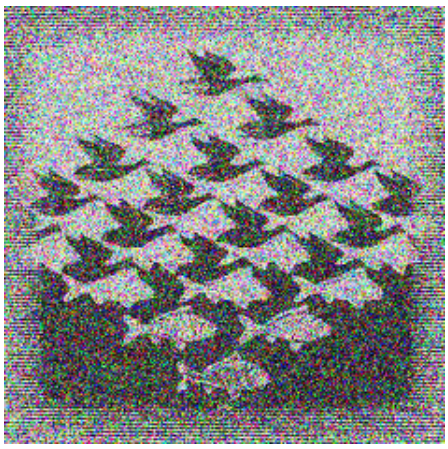
```c
#include <math.h>
#include <stdlib.h>
#include "./a4.h"

#define data(row, col) data[row * width + col]

void mutate(Individual *individual, double rate) {
  PPM_IMAGE image = individual->image;
  PIXEL *data = image.data;
  int height = image.height, width = image.width;
  int max_color = image.max_color;

  int n = width * height, pixToChange = (int)(rate / 100 * n);

  for (int i = 0; i < pixToChange; i++) {
    int row = rand() % height;
    int col = rand() % width;

    data(row, col).r = rand() % (max_color + 1);
    data(row, col).g = rand() % (max_color + 1);
    data(row, col).b = rand() % (max_color + 1);
  }
}

void mutate_population(Individual *individual, int population_size,
                       double rate) {
  // Only mutating from pop/4 to pop-1
  int i = (int)population_size / 4;
  for (i; i < population_size; i++) {
    mutate(individual + i, rate);
  }
}
```

Output Images



```
> make escher
```

```
> make mcmaster
```



This image was produced using 1.5 million generations with population size 160 and rate 4e-2. It took approximately 6 hours to run using multithreaded versions of `comp_fitness_population` and `generate_population`.

Valgrind Output

```
caleb:SE-2S03-A4$ valgrind ./evolve me.ppm me2.ppm 10 24 3e-2
==954== Memcheck, a memory error detector
==954== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==954== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==954== Command: ./evolve me.ppm me2.ppm 10 24 3e-2
==954==
==954== error calling PR_SET_PTRACER, vgdb might block
==954==
==954== HEAP SUMMARY:
==954==     in use at exit: 0 bytes in 0 blocks
==954==   total heap usage: 53 allocs, 53 frees, 4,202,488 bytes allocated
==954==
==954== All heap blocks were freed -- no leaks are possible
==954==
==954== For counts of detected and suppressed errors, rerun with: -v
==954== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

gprof Optimization

```
[mechc2@moore ~/SE-2S03-A4-master] gprof ./evolve
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 88.34      5.49     5.49    24000     0.00     0.00  comp_distance
 11.28      6.20     0.70     5994     0.00     0.00  recombine
  0.48      6.23     0.03       24     0.00     0.00  generate_random_image
  0.00      6.23     0.00    17982     0.00     0.00  mutate
  0.00      6.23     0.00     1000     0.00     0.01  comp_fitness_population
  0.00      6.23     0.00      999     0.00     0.00  crossover
  0.00      6.23     0.00      999     0.00     0.00  mutate_population
  0.00      6.23     0.00        2     0.00     0.00  free_image
  0.00      6.23     0.00        1     0.00     6.23  evolve_image
  0.00      6.23     0.00        1     0.00     0.03  generate_population
  0.00      6.23     0.00        1     0.00     0.00  read_ppm
  0.00      6.23     0.00        1     0.00     0.00  write_ppm

[...]
```

The gprof output of our evolve program showed that the most time consuming process was computing distance for the fitness values. This makes sense when considering every pixel of the image requires a distance calculation. To improve the performance of this function we could use multithreading so that the calculations may be performed in parallel. We would spawn threads at the level of `comp_fitness_population` by splitting up the population into subpopulations. This would benefit performance specifically over multithreading at the `comp_fitness` level because it only requires threads to be spawned once a generation. We decided to implement multithreading for experimentation and the gprof results showed that the function used a lower percentage of the runtime. Furthermore this trend continued with an increase population size.
Another potential method to gain performance would be to use a single instead of double to store the fitness values. This would cause the mathematical operations in `comp_fitness` to be performed quicker with potentially negligable effect on the precision of the algorithm.

Visualizing Image Evolution

The submitted video should be playable in VLC and is also available at youtu.be/R7ZCM8FoQk0

As the image evolves the rate of change decreases. For the first 5,000-7,500 generations a lot of change can be seen. However the closer the evolved image gets to its goal, the slower the rate of change becomes. Once 10,000 or more generations have been simulated only a few pixels will be seen changing. After being plotted on a graph a trendline can mostly easily be made using the logarithm function.