SE 2S03 ASSIGNMENT 4

Due date: 28 November, by 12:40 in class.

- This assignment is out of 30. With the bonus points, you can obtain 38/30.
- You can work individually or in a group of two. Each group member receives the same grade.
- Students who want to work in groups of two should enter their names in one row in the spreadsheet
  https://docs.google.com/spreadsheets/d/1yN7PVya292D-Ilf2l-EHS3hbBInCgXCXfeMgjCLQI2M/edit?usp=sharing
- Groups should be finalized before November 18, and any changes after that will not be considered. There should not be any duplicated names.
- Students working individually do not have to enter their names in the spreadsheet.

1. Introduction

In a genetic algorithm, a population of individuals evolves over time. The stronger individuals reproduce, and the weaker ones die. The surviving ones can also mutate. With each individual, there is a fitness number that indicates how strong that individual is. For more details, see https://en.wikipedia.org/wiki/Genetic_algorithm. A genetic algorithm terminates when a given number of iterations is reached or when a solution is found, that is, an individual with certain fitness.

The goal of this assignment is to compute an image, using a basic genetic algorithm approach, that is close to a given image.

Download and unzip http://www.cas.mcmaster.ca/~nedialk/COURSES/2s03/private/a4.zip

2. Data types and functions, file `a4.h`

Use the following file (and do not change it)

```
#ifndef INCLUDED_A4_H
#define INCLUDED_A4_H
typedef struct { unsigned char r, g, b; } PIXEL;
typedef struct {
  PIXEL *data;        // pointer to an array of width*height PIXELs
  int width, height; // width and height of image
  int max_color;      // largest value of a color
} PPM_IMAGE;
typedef struct {
  PPM_IMAGE image; // image
  double fitness;  // fitness
} Individual;
```

Date: 14 November, 2018.

```c
// Read and write PPM file
PPM_IMAGE *read_ppm(const char *file_name);
void write_ppm(const char *file_name, const PPM_IMAGE *image);
// Random image and population
PIXEL *generate_random_image(int width, int height, int max_color);
Individual *generate_population(int population_size, int width, int
   height, int max_color);
// Fitness
double comp_distance(const PIXEL *A, const PIXEL *B, int image_size)
   ;
void comp_fitness_population(const PIXEL *image, Individual *
   individual, int population_size);
// Crossover
void crossover(Individual *individual, int population_size);
// Mutation
void mutate(Individual *individual, double rate);
void mutate_population(Individual *individual, int population_size,
   double rate);
// Compute image
PPM_IMAGE *evolve_image(const PPM_IMAGE *image, int num_generations,
    int population_size, double rate);
// Free image
void free_image(PPM_IMAGE *p);
#endif
```

3. Implementation (25 points)

Implement the bodies of the functions that follow and store them in files with names as indicated at the heading of each subsection. Each of your files must include `a4.h`.

3.1. Input/output, file `readwriteppm.c`.

The input image is in a PPM P3 format, and your program should produce an image in the same format. See https://en.wikipedia.org/wiki/Netpbm_format.

```c
PPM_IMAGE *read_ppm(const char *file_name);
```

Reads a PPM image file and returns a pointer to a `PPM_IMAGE` structure containing the pixels of the image, width, height, and max color.

```c
void write_ppm(const char *file_name, const PPM_IMAGE *image);
```

Writes a PPM image into a file in PPM format.

3.2. Generating a population, file `population.c`.

```c
PIXEL *generate_random_image(int width, int height, int
   max_color);
```

Returns a pointer to an array of `width*height` pixels. The red, green, and blue values of a pixel are set randomly, where each value is `>= 0` and `<= max_color`.

```
Individual *generate_population(int population_size, int width,
    int height, int max_color);
```

Returns a pointer to an array of size `population_size`. Each entry is a structure of type `Individual`. For each entry in this array, the `PPM_IMAGE` structure should be properly initialized.

- The image should be generated using `generate_random_image`.
- The `width`, `height` and `max_color` should be initialized.

3.3. **Computing fitness, file `fitness.c`.** Denote the input image by A and an arbitrary image by B. Let n be the number of pixels in each of these images.

Denote a pixel i in A by $A(i)$ and in B by $B(i)$. Let

$$d(i) = [A(i).r - B(i).r]^2 + [A(i).g - B(i).g]^2 + [A(i).b - B(i).b]^2$$

where r, g, b denote the values for red, green, and blue. The distance between A and B is

$$f(A, B) = \sqrt{\sum_{i=1}^{n} d(i)}. \tag{1}$$

```
double comp_distance(const PIXEL *A, const PIXEL *B, int
   image_size);
```

This function computes the distance (1) between the images at `A` and `B`, where each is of size `image_size`.

```
void comp_fitness_population(const PIXEL *image, Individual *
   individual, int population_size);
```

Computes the fitness of each individual in the population. The fitness of `individual[i]` is the distance between the input `image` and `individual[i].image.data`. `individual` is a pointer to an array of size `population_size`.

3.4. **Crossover.** You can use `crossover.c` from `a4.zip`.

3.5. **Mutation, file `mutate.c`.**

```
void mutate(Individual *individual, double rate);
```

`rate` is the percentage of pixels to be mutated. For an image of size `n = width*height`, this function selects `(int)(rate/100*n)` pixels at random and sets random values for the RGB colors of these pixels.

```
void mutate_population(Individual *individual, int
   population_size, double rate);
```

This function mutates all individuals starting from index `population_size/4` to the end of the population. That is, from `individual + population_size/4`.

3.6. Evolving an image, file `evolve.c`.

```
PPM_IMAGE *evolve_image(const PPM_IMAGE *image, int
   num_generations, int population_size, double rate);
```

This functions takes as input a pointer to `PPM_IMAGE`, number of generations, population size, and mutation rate, and performs the following steps.

(1) Generate a random population of `population_size`.
(2) Compute the fitness of each individual.
(3) Sort the individuals in non-decreasing order of fitness. The first individual will have the closest distance to the original image, and it is the most "fitted".
(4) For generation 1 to `number_generations`
   (a) do a crossover on the population
   (b) mutate individuals from `population_size/4` to `population_size-1`.
   (c) compute the fitness of each individual
   (d) sort the individuals in non-decreasing order of fitness value
(5) Return a pointer to a `PPM_IMAGE` containing the fittest image, that is, the one with the smallest fitness value.

```
void free_image(PPM_IMAGE *p);
```

Releases the memory associated with the image at łp.

3.7. Main program. Use the main program from `a4.zip`.

3.8. Makefile. Use the makefile from `a4.zip`.

4. Performance (5 points)

4.1. Profiling (1 point). Profile the execution of `evolve` using `gprof`. Discuss the computationally most expensive parts of your code. In the printed output of `gprof` highlight the 5 lines of code that contribute the most to the execution time. Explain why they contribute the most.

4.2. Code optimization (4 points). Based on the `gprof` output, can you improve the performance of your code? If so, describe how you would have done it and provide sufficient detail showing your improvements.

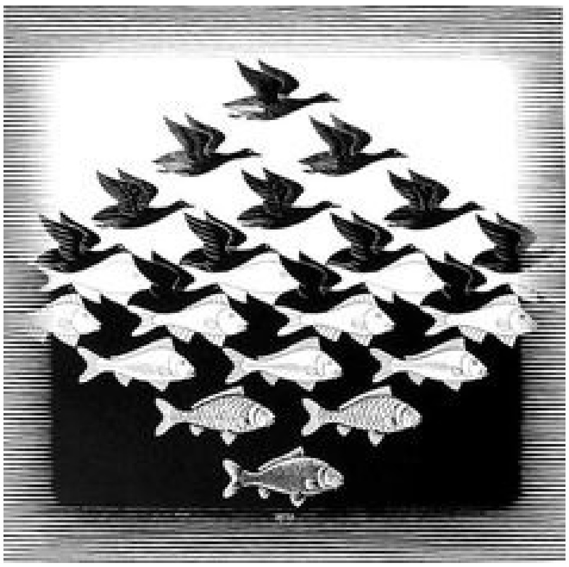
5. Bonus (8 points)

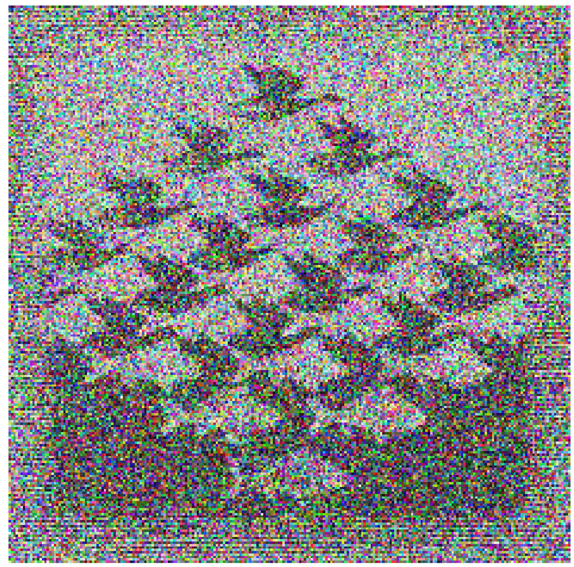
5.1. Better algorithm (5 points). Play with various input parameters. Read about genetic algorithms and see if you can produce better evolve, crossover, and mutate functions so that you can obtain better images with smaller number of generations and population size.

5.2. **Visualizing the image evolution (3 points).** You can output each image into a file. Then you can stitch them together into a video. If you obtain a good result, this is effective for your resume, especially if you have the animation on YouTube. To convert to jpg, or other formats, you can use the `convert` utility.

6. Examples

Below on the left is an input image `me.ppm` and on the right is the image `me2.ppm` produced by `make escher`.





Below on the left is an input image `mcmaster.ppm` and on the right is the image `mcmaster2.ppm` produced by `make mcmaster`.

7. Submit

Group of two: only one student should submit.

- SVN under directory `A4`:
  (1) The files `readwriteppm.c`, `fitness.c`, `population.c`, `evolve.c` and `mutate.c`.
  (2) The images from `make escher` and `make mcmaster`.
  (3) Video file, if you produce such.
- Hard copy containing

(1) `readwriteppm.c`, `fitness.c`, `population.c`, `evolve.c` and `mutate.c`.
(2) The images from `make escher` and `make mcmaster`.
(3) The output of `valgrind` on one of your executions. (Take smaller numbers for population size and number of generations, as otherwise the execution will take a long time.) To obtain full marks for the implementation, `valgrind` must not report any problems.
(4) The `gprof` output and your discussion on optimization.
(5) In 5.1, if you can produce better images than the two in this pdf:
   - the parameters that you have used
   - short descriptions of your functions, if they are different from the required in this assignment.
(6) If you did 5.2, write a discussion on how the image is evolving over time, and what are the changes that you notice with each iteration.