The Optional Final Project

Genetic Optimization Algorithm in C and Generating a RGB Image

(25 points based on the second marking scheme mentioned in course outline **+4** points bonus)

Pedram Pasandide

Due Date: 22 December 2023

Important Notes:

- This is an **optional** project. So **it is up to you** if you want to do it or not. In this course, we have two grading system. One with (Option 1) and the other one without a final project (Option 2) (Please refer to the course outline). If some students chose to do the final project (**Option 2**), then what we do is:

  1. Including the final project according to the marking scheme "with" final project (**Option 2**).
  2. Using the marking scheme "without" a final project (**Option 1**).

  And then we just use the better of the two grades! So you don't need to be worried about the final grade!

- This project is about optimization using Genetic Algorithm. If you produce good results, this will be one of the best projects that you are going to do during your undergraduate studies.

- This project would be suitable for those students who did good in the second assignment, or they would like to continue working on optimization algorithms.

1 Introduction

We skip the part about introduction since we had this in Second Assignment. The goal of this assignment is to compute an image, using a basic genetic algorithm approach, that is close to a given image. The given images are in `PPM` format (files with `.ppm` extension).

Files you have downloaded are:

- `main.c`: This is were you call all the functions. Follow the comments.

- `functions.h`: This were we declared all the function, but the real implementations are in source files. This how it looks like:

```c
#ifndef INCLUDED_functions_H
#define INCLUDED_functions_H

////////////////////////////////////////////////////////////////////
// DO NOT CHANGE anything in this part
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;

// 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 Objective_function(const PIXEL *A, const PIXEL *B, int image_size);
void compute_objective_function(const PIXEL *image, Individual *individual, int
    POPULATION_SIZE);

// Free image
void free_image(PPM_IMAGE *p);

int generate_random_int(const int lbound, const int ubound);

////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////
// You can make changes this part if you have really a good explanation

// Compute image
PPM_IMAGE *evolve_image(const PPM_IMAGE *image, int num_generations, int POPULATION_SIZE
    , double rate);

// Crossover
void crossover(Individual *individual, int POPULATION_SIZE);

// Mutation
void mutate(Individual *individual, int POPULATION_SIZE, double rate);
////////////////////////////////////////////////////////////////////

#endif
```

- `readwriteppm.c`: The source code for reading and write a `PPM` image.
- `population.c`: Generating initial and random population happens here.
- `fitness.c`: Computing the objective function.
- `crossover.c`: Crossover function.
- `mutate.c`: Mutation function.
- `evolve.c`: The whole GA is here.
- `Pedram_Convert.ipynb`: The Python code visualizing the results.
- and a bunch images with `PPM` format.

2 Implementing the Code

2.1 Input/output, Read and Write an image with `PPM` format (5 points)

The input image (objective) is a `PPM` format image, and you program should be able to produce the same format of output with the same image size. Visit Netpbm for more information about this format.

```
PPM_IMAGE *read_ppm(const char *filename);
```

This function reads a `PPM` image file and returns a pointer to a `PPM_IMAGE` structure containing an array of pixels in the image, width, height, and max colour.

The structure `PPM_IMAGE` is defined in `functions.h`:

```
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;
```

Every pixel has three values for red, green, and blue ranging from 0 to 255. That's why the type `PIXEL` is a structure we defined as:

```
typedef struct { unsigned char r, g, b; } PIXEL;
```

The following function writes a `PPM_IMAGE` structure image into a file in `PPM` format.

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

I already gave you enough files (4 images) with `.ppm` format to test your program. But if you want in the future to have more `PPM` files you can use the **Python** code (`Convert_image_to_ppm.ipynb`) the following part to convert any format of images and save them into `PPM` format. Make the necessary changes to the code to convert your images into a format you want. *You have to change the names based on your needs.*

2.2 Develop the code for Genetic Algorithm (10 points)

This we want to have a more well-structured code. In `main.c`, we **only** call the functions, and we will NOT have any initializing variables, allocating the memory, or having any iterations.

1. First, we need to generating an initial population using the function `generate_population()` with the implementation in file `population.c`.

```
Individual *generate_population(int POPULATION_SIZE, int width, int height, int
    max_color)
```

This function will return `population`, where every element of `population` will have a structure of `Individual`. We can say a `Individual` is a guess or a set of decision variables, and every set of decision variable will have a fitness value. So every `Individual` holds a value for `fitness` and an `image` with the structure of `PPM_IMAGE`. Later on, every `image` will be sent to objective function and the value of `fitness` for each individual image (chromosome or set of decision variables) will be computed. So in this function we need to create a `population` with the type `Individual`, like:

```
Individual *population = malloc(POPULATION_SIZE * sizeof(Individual));
```

Each image (each row) in the `population`, will be created using a function called:

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

This function returns a pointer to an array of `width*height` size where every element has a type of `PIXEL`. The red, green, and blue values of a pixel are set randomly, where each value is $>= 0$ and $<=$ `max_color`. **Use** `generate_random_int()` to get this random number.

2. Now we need to compute the fitness value, file `fitness.c`, for each individual in `population`.

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

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

```
double Objective_function(const PIXEL *A, const PIXEL *B, int n)
```

This function computes the `fitness` between the images at A and B, where each has a size of $n = width \times height$.

The function `compute_objective_function()`, computes the fitness of each individual in the population (`population[i].fitness`).

```
void compute_objective_function(const PIXEL *goal, Individual *population,
int POPULATION_SIZE)
```

The fitness will be computed calling `Objective_function` and sending `population[i].image.data` for each `population[i].fitness`.

3. You can use the file `crossover.c` and there is not need to change it. Here we don't need a crossover rate.

4. Mutation happens in `mutate.c` and calling the function:

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

`rate` is the mutation rate (MR) given by the use a value from 0 to one. For an image of size $n = width \times height$, this function selects `(int)(rate*n)` pixels of each individual at random and sets random values for the RGB colours of these pixels.

This function mutates all individuals starting from index `POPULATION_SIZE/4` to the end of the population. This approach makes sure that the first 25% of individuals will remain untouched.

5. Evolving the images using:

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

implemented in file `evolve.c`.

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". For this part you can use the function `qsort()` in C.
4. For generation 1 to `number_generations`
   (a) Do a crossover on the population
   (b) Mutate individuals from `POPULATION_SIZE/4` to the end of the population. This makes sure **the top and strongest 25%** individuals will not be mutated.
   (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.

At the end I have to be able to run your code with the following format:

```
./GA images1.ppm output_image.ppm 10 500 3e-4
```

where `GA` is the name of program after compiling the code with your `Makefile`, `images1.ppm` is the objective image that we want to recreate, `output_image.ppm` is the created image using optimization, `10` is the maximum number of generations, `500` is the population size and `3e-4` is the mutation rate. This is `images1.ppm`:

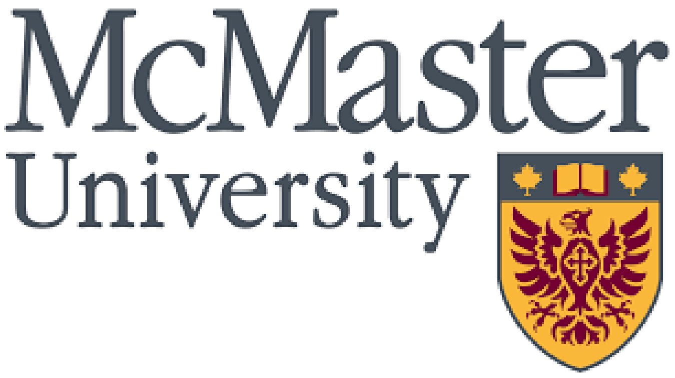

Figure 1: The original `images1.ppm`

and I want to create `output_image.ppm` which is close to this picture. This the output in my computer:

```
Reading the image ...

File images1.ppm:
302x167, max color 255

Optimization Specifications:
Max Generation = 10
Population size = 500
Mutation Rate = 3.00e-04
Pixels to mutate 15

The Genetic Algorithm is started ...
generation     0 fitness 5.218277e+04  change from prev 1.50e-01%
generation     1 fitness 5.218277e+04  change from prev -0.00e+00%
generation     2 fitness 5.216223e+04  change from prev 3.94e-02%
generation     3 fitness 5.213109e+04  change from prev 5.97e-02%
generation     4 fitness 5.212933e+04  change from prev 3.37e-03%
generation     5 fitness 5.212545e+04  change from prev 7.44e-03%
generation     6 fitness 5.208937e+04  change from prev 6.93e-02%
generation     7 fitness 5.205929e+04  change from prev 5.78e-02%
generation     8 fitness 5.204672e+04  change from prev 2.42e-02%
generation     9 fitness 5.203634e+04  change from prev 1.99e-02%
CPU Time  1.872312 seconds
```

But when I am sure my algorithm is already working, I don't need to do the print for every iteration. Let's say for the inputs of:

`./GA images1.ppm output_image.ppm 10000 500 3e-4`

I have:

```
Reading the image ...

File images1.ppm:
302x167, max color 255

Optimization Specifications:
Max Generation = 10000
Population size = 500
Mutation Rate = 3.00e-04
```

```
Pixels to mutate 15

The Genetic Algorithm is started ...
Fitness for initial guess = 5.223731e+04
After generation 10000, fitness = 4.004365e+04, change from initial= -23.31%
CPU Time  632.024249 seconds
```

2.3 Profiling and Code Optimization (5 points)

If you don't analyze your code with profiling tools your program doesn't mean anything! You have to always make sure that your program has the best performance. Use `gprof` or `VTune`. In the second assignment you studied about Genetic Algorithm. Try to improve your program, like crossover, mutation, evolve, sorting function, and make it work faster. In this case, improving GA optimization means obtaining better images with lest number of generations and population size. Write your results in a table like:

Table 1: The **Best** Results for:

Image			Pop Size	Max Gen	Fitness	MR	CPU time (Sec)
name	width	height					
images1.ppm	302	167	500	10000	4.00e+04	3e-4	632.02
images2.ppm							
images3.ppm							
images4.ppm							

Include the final pictures you created for **all images**. For example for image `images1.ppm`, this is the final result (`output_image.ppm`) after `10000` generations:

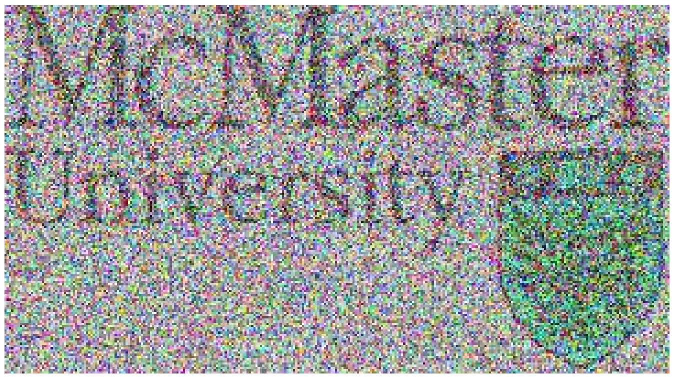
Figure 2: GA optimization for `images1.ppm` with 500 population size, 10000 generations, and mutation rate of 3e-4

2.4 Visualizing the Image Evolution (5 points)

In every generation, on individual in the population has the lowest fitness value. You can save the best individual, lets say every 200 generations, and stitch them together into a video. Upload the video on your YouTube, like My Result, and **beautifully** present your result in the description. Like your are selling your program. Also in the description include the link to your codes. To stick the pictures you created you can use the **Python** code `Convert_images_to_mp4.ipynb`. First I saved the best images every 200 generations, and I save them with the format `pedram_*.ppm` in the same directory. For example, `pedram_0.ppm`, `pedram_200.ppm`, `pedram_400.ppm`, `pedram_600.ppm`, ... `pedram_10000.ppm`, are images after 0, 200, 400, 600, and 10000 (the last output file) generation, respectively.

Then, move all these images into the directory where you have the Python code, and stich all the files with the name `pedram_*.ppm` together to create a `.mp4` format video, with frames per second of `fps=2.0`, and the resolution of `(302, 167)`. The values `(302, 167)` must be equal the width and height of the images which for `images1.ppm` it was `(302, 167)`. The file with name `output_video.mp4` is the output video of the **Python** code. *You have to change the names based on your needs.*

2.5 Parallel Computing - Bonus (+4 points)

The algorithm is slightly different from the second assignment, and the main reason behind it is to avoid using condition statements as much as possible. The both projects are about Genetic Algorithm, however, one with many `if` statements and making it almost pointless to do parallel computing. But in this project we avoided using any condition statement. Implement the parallel computing using OpenMP by including `omp.h` and provide the following tables but with **wall clock (Time)**:

Table 2: The **Best** Results with Parallel Computing **with 2 Threads**.

Image			Pop Size	Max Gen	Fitness	MR	Time (Sec)
name	width	height					
images1.ppm							
images2.ppm							
images3.ppm							
images4.ppm							

Where Threads is the number of threads given by the user when running the code, like

```
./GA images1.ppm output_image.ppm 10 500 3e-4 6
```

The last input `6` shows the number of threads used.

Table 3: The **Best** Results with Parallel Computing **with 4 Threads**.

Image			Pop Size	Max Gen	Fitness	MR	Time (Sec)
name	width	height					
images1.ppm							
images2.ppm							
images3.ppm							
images4.ppm							

Table 4: The **Best** Results with Parallel Computing with **8 Threads**.

Image			Pop Size	Max Gen	Fitness	MR	Time (Sec)
name	width	height					
images1.ppm							
images2.ppm							
images3.ppm							
images4.ppm							

3 Report and `Makefile`

Provide a `Makefile` to compile your code, and in the report with details tells us how your `Makefile` works. Your report **must** be in `README.tex` file. Other formats will not be accepted. Create the `README.pdf` file from `README.tex` and submit both formats. You can use LaTex file format that I have sent you.

4 Submission

Submit On Avenue following files:

- `main.c`, do NOT implement any function here. Here you should just call other functions.
- `functions.h`, if you changed anything here let me know and tell me why.
- `readwriteppm.c` This is the first step, check many times it is correct.
- `fitness.c`.
- `crossover.c`, if you modified this explain why.
- `mutate.c`, if you implemented a different algorithm explain why and the improvements.
- `evolve.c`, if you implemented a different algorithm explain why and the improvements.
- `README.pdf` and `README.tex`, a detailed and perfect description!

- `Makefile`.