Jatan Bhatt, Ridhwan Chowdhary
Friday November 30, 2018
2S03

Assignment 4

1)

readwriteppm.c

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

PPM_IMAGE *read_ppm(const char *file_name)
{

   FILE *file = fopen(file_name,"r");
   if (!file)
   {
      printf("File does not exist\n");
      return NULL;
   }

   PPM_IMAGE *image;
   image = malloc(sizeof(PPM_IMAGE));

   char type[100];
   fscanf(file,"%s",type);
   if (strcmp(type,"P3") != 0)
   {
      printf("Not a PPM file\n");
      exit(1);
   }
   fscanf(file,"%d %d %d",&image->width,&image->height,&image->max_color);
   printf("Width is: %d, Height is: %d, Max color is: %d\n",image->width,image->height,image-
>max_color);

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

   char red;
   char green;
   char blue;

   for (int i = 0; i < total_pixels ; i++)
   {
      fscanf(file,"%hhu %hhu %hhu",&red,&green,&blue);
      image->data[i].r = red;
      image->data[i].g = green;
```

```
        image->data[i].b = blue;
    }
    fclose(file);
    return image;

}

void write_ppm(const char *file_name, const PPM_IMAGE *image)
{
    FILE *file = fopen(file_name,"w");
    if (!file)
    {
        printf("File could not be created\n");
        exit(1);
    }

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

    int dimension = image->width*image->height;

    for (int i = 0; i < dimension; i++)
    {
        fprintf(file,"%d ",image->data[i].r);
        fprintf(file,"%d ",image->data[i].g);
        fprintf(file,"%d ",image->data[i].b);
        if (i>0 && i%image->width == 0) fprintf(file,"\n");
    }
    fclose(file);
}
```

fitness.c

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

double comp_distance(const PIXEL *A, const PIXEL *B, int image_size)
{
    long double distance = 0;
    for (int i = 0; i < image_size; i++)
    {
        distance += (A[i].r - B[i].r)*(A[i].r - B[i].r) + (A[i].b - B[i].b)*(A[i].b - B[i].b) + (A[i].g -
B[i].g)*(A[i].g - B[i].g);
    }
    return sqrt(distance);
}

void comp_fitness_population(const PIXEL *image, Individual *individual, int population_size)
{
    for (int i = 0; i < population_size; i++)
```

```
    {
        int total_pixels = individual[i].image.width*individual[i].image.height;
        individual[i].fitness = comp_distance(image,individual[i].image.data,total_pixels);
    }
}
```

population.c

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

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

    for (int i = 0; i < total_pixels; i++)
    {
        pixels[i].r = rand()%(max_color+1);
        pixels[i].g = rand()%(max_color+1);
        pixels[i].b = rand()%(max_color+1);
    }
    return pixels;
}

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

    for (int i = 0; i < population_size; i++)
    {
        PPM_IMAGE image;
        image.width = width;
        image.height = height;
        image.max_color = max_color;
        image.data = generate_random_image(width, height, max_color);
        population[i].image = image;
        population[i].fitness = 0;
    }
    return population;
}
```

evolve.c

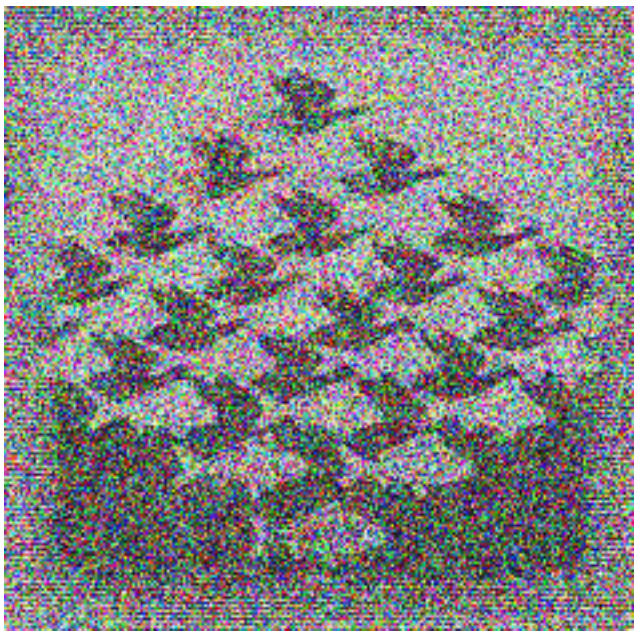
```
#include "a4.h"
#include <stdio.h>
#include <stdlib.h>

int compare (const void * a, const void * b)
{

  Individual *A = (Individual *)a;
  Individual *B = (Individual *)b;

  return ( A->fitness - B->fitness );
}

PPM_IMAGE *evolve_image(const PPM_IMAGE *image, int num_generations, int
population_size, double rate)
{
    Individual *random = generate_population(population_size, image->width, image->height,
image->max_color);
    //comp_fitness_population(image->data, random, population_size); //commented out first
run, added to for loop
    //qsort(random, population_size, sizeof(Individual), compare);

    for (int i = 0; i < num_generations; i++)
    {
        crossover(random, population_size);
        mutate_population(random, population_size, rate);
        comp_fitness_population(image->data, random, population_size);
        qsort(random,population_size,sizeof(Individual),compare);

        printf("Gen: %d\tFit: %f\n", i, random[0].fitness); //for debugging
        printf("\n");
    }

    for (int j = 1; j < population_size; j++)
    {
        free(random[j].image.data);
    }
    return &random->image;
}

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

mutate.c

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

void mutate(Individual *individual, double rate)
{
   int total_pixels = individual->image.width * individual->image.height;
   int num_random = (rate / 100) * total_pixels;
   int r_index;

   for (int i = 0; i < num_random; i++)
   {
      r_index = rand()%(total_pixels+1);

      individual->image.data[r_index].r = rand()%256;
      individual->image.data[r_index].g = rand()%256;
      individual->image.data[r_index].b = rand()%256;
   }
}

void mutate_population(Individual *individual, int population_size, double rate)
{
   Individual *curr = individual + population_size/4;

   for (int i = population_size/4; i < population_size; i++)
   {
      mutate(curr, rate);
      curr++;
   }
}
```

2)

make escher output:



make mcmaster output:

3) **valgrind** output: (small population size and small number of generations)

```
==18873==
==18873== HEAP SUMMARY:
==18873==     in use at exit: 0 bytes in 0 blocks
==18873==   total heap usage: 13 allocs, 13 frees, 1,492,464 bytes allocated
==18873==
==18873== All heap blocks were freed -- no leaks are possible
==18873==
```

4) **gprof** output after make escher:

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 76.11    124.51   124.51  1200000     0.00     0.00  comp_distance
 23.99    163.76    39.25   300000     0.00     0.00  recombine
  0.12    163.96     0.20   900000     0.00     0.00  mutate
  0.01    163.98     0.02    50000     0.00     0.00  comp_fitness_population
  0.01    164.00     0.02       24     0.00     0.00  generate_random_image
  0.01    164.01     0.01    50000     0.00     0.00  mutate_population
  0.00    164.01     0.00    50000     0.00     0.00  crossover
  0.00    164.01     0.00        2     0.00     0.00  free_image
  0.00    164.01     0.00        1     0.00   164.01  evolve_image
  0.00    164.01     0.00        1     0.00     0.02  generate_population
  0.00    164.01     0.00        1     0.00     0.00  read_ppm
  0.00    164.01     0.00        1     0.00     0.00  write_ppm
```

The five most computationally expensive functions:

1. comp_distance
2. recombine
3. mutate
4. comp_fitness_population
5. generate_random_image

Based on the gprof output, it is evident that comp_distance and recombine represent the bulk of execution time.

The reason for this is that comp_distance must be called to calculate the fitness of each individual, every generation, for x amount of “people”. comp_distance also takes longer to execute every call as it is performing a mathematically intensive function.

Recombine also requires a lot of time for execution as it must “stitch” together images in the random population.

The sorting algorithm surprisingly does not take a significant percentage of time, likely due to the implementation of qsort and the fact that other functions require a lot of time.

One way to reduce run time would be to figure out a quicker way to compute fitness, rather than manually computing the “distance” between each pixel. For example, maybe every other pixel could be compared, giving a less accurate final image but almost halving the time required to compute. The distance formula is also used for other applications, such as finding the difference between two 3D vectors. An improvement would be to express each pixel as a vector and implement a library which is optimized to compute the distance between vectors.