Assignment 3: Developing Genetic Optimization Algorithm in C with case study

Edris Lutfi

1 Introduction

In this file, I will explain how to run and compile my code for the assignment, how it works, and my findings. To analyze my codes, I have listed `functions.c`, `GA.c`, and `Makefile` in the appendix. I have not listed any of the other codes, as it is not required to submit.

2 Explanation

2.1 Explanation for `functions.c`

The purpose of this code is to create functions that other files (specifically GA.c) can pull from. Here we have many functions. 'The generate random' function is used to generate a random number between a given min and max value. The generate int function is similar, however it generates a random integer between the min and max. The generate population function randomly initializes all the values in 'population[i][j] so it is easy to put values in there later on. The compute objective function is used to compute 'fitness[i]' for each set of decision variables. Within this function, the computation works by pulling from the specific 'objective function' from the `OF.c` code that was given to us. This is where the specific functions (like levy) is calculated. The crossover function combines traits from two parents to create offspring to improve the population's overall quality by introducing genetic diversity (think of evolution). This code selects parents based on their fitness, then it swaps parts of their genes at random, and then it creates new offspring with the mixed traits (increasing the chances of better solutions). Lastly, the mutations function makes random changes to the populations individuals by altering the genes based on a given mutation rate. This maintains genertic diversity and prevents the algorithm from being stuck at a local optima. It then replaces certain genes with random values, and then inputing these changes back to the actual popualtion.

2.2 Explanation for `GA.c`

This code file runs the genetic algorithm to find the best solution through optimization (finding the lowest best solution that the code is capable to find with the specific parametres). It also uses the functions in the `functions.c` file to do this. Then, when it finds the best solution, it prints the results and the CPU time used (so that we can compare the speed as well).

2.3 Explanation for `Makefile`

The Makefile file has systems created to execute the commands of compiling and for removing object files and executables when needed. First off, to compile the GA (genetic algorithm), the functions.c file, GA.c file, and OF.c file has to be compiled into seperate object files (eg. GA.o, etc.), then all of the object files need to be compiled and linked together. Thererfore, instead of doing all of that seperately, I have created a Makefile so that when you write "make" in the terminal, it automatically does all the compilation for you. It also automatically compiles with flags so that it could catch any errors and so that we can use some mathematical shortcuts (with -lm). In the Makefile there are also some shortcuts that I have created so that the code is more efficient. Finally, there is also a clean rule to remove object files and executables, so that you could remove all object files in one go.

3 Results

The output should look similar to the following results in your terminal

```
Genetic Algorithm Parameters:
Population Size: 1000
Max Generations: 100
Crossover Rate: 0.50
Mutation Rate: 0.20
Stop Criteria: 0.00
CPU time: 0.058512 seconds
Best solution found:
x[0] = 2.845825
x[1] = -0.284620
x[2] = 1.308549
x[3] = 1.311407
x[4] = 1.599055
x[5] = 0.814461
x[6] = -1.686367
x[7] = -1.710302
x[8] = 4.978297
x[9] = -1.396060
Objective function value: 0.000045

Best fitness: 4.505144e-05
```

The following tables are the results of the trials given in the Assignment.

Table 1: `NUM_VARIABLES = 10`, Crossover Rate = 0.5, Mutation Rate = **0.05** for **Levy** function

Pop Size	Max Gen	Best Fitness	CPU time (Sec)
10	100	2.727469e+00	0.000915
100	100	2.048704e-01	0.009966
1000	100	6.991386e-03	0.094975
10000	100	9.377000e-05	3.549659
1000	1000	6.943139e-06	0.876949
1000	10000	3.366370e-08	8.680238
1000	100000	4.401841e-08	11.772110
1000	1000000	1.013187e-08	12.256150

Table 2: `NUM_VARIABLES = 10`, Crossover Rate = 0.5, Mutation Rate = **0.2** for **Levy** function

Pop Size	Max Gen	Best Solution	CPU time (Sec)
10	100	7.068118e+00	0.001524
100	100	7.625654e-01	0.008468
1000	100	1.632830e-02	0.098179
10000	100	1.008243e-04	3.559379
1000	1000	1.345167e-05	0.920098
1000	10000	1.399565e-07	8.963974
1000	100000	5.649642e-08	19.223536
1000	1000000	1.805117e-08	20.628641

Table 3: `NUM_VARIABLES = 50`, Crossover Rate = 0.5, Mutation Rate = **0.2** for **Levy** function

Pop Size	Max Gen	Best Solution	CPU time (Sec)
10	100	7.026583e+03	0.003881
100	100	5.793629e+03	0.037687
1000	100	4.948267e+03	0.351246
10000	100	4.703293e+03	6.062733
1000	1000	4.500193e+03	3.382423
1000	10000	3.936763e+03	17.067616
1000	100000	4.642693e+03	13.155547
1000	1000000	4.131059e+03	18.439497

Table 4: `NUM_VARIABLES = 10`, Crossover Rate = CR, Mutation Rate = MR for **all** function

Function	Pop Size	Max Gen	CR	MR	Best Solution	CPU time (Sec)
Griewank	1000	100	0.5	0.2	9.010000e+02	0.076520
Levy	1000	100	0.5	0.2	5.755090e-03	0.099216
Rastrigin	1000	100	0.5	0.2	2.892471e+02	0.093854
Schwefel	1000	100	0.5	0.2	1.060515e+03	0.082025
Trid	1000	100	0.5	0.2	2.756436e+00	0.061472
Dixon-Price	1000	100	0.5	0.2	8.888295e-01	0.076225
Michalewicz	1000	100	0.5	0.2	-3.902798e+00	0.089732
Powell	1000	100	0.5	0.2	2.298443e-06	0.060065
Styblinski-Tang	1000	100	0.5	0.2	-2.862179e+02	0.087267

4 Appendix

In this section, I have listed three code files that are required to submit. I have not included the `OF.c` and the `functions.h` files because those were given and have not been changed. Take note that I have left comments in these files and some of the comments are placed there to compare different ways to write the same code (as I have improved certain aspects of the code, but I wanted to leave the original one there to compare with). `functions.c`

```
  // Your CODE: Include everything necessary here
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "functions.h"


double generate_random(double min, double max)
{
    // Your CODE: implement a function to return a value between
       min and max
    if (min > max)
    {
        return -1; // Error: invalid range
    }

    return min + (double)rand() / RAND_MAX * (max - min);
}

int generate_int(int min, int max)
{
    // Your CODE: implement the function to return a random integer
        value

    int int_min = (int)round(min);
    int int_max = (int)round(max);

    // Ensure the range is valid
    if (int_min > int_max)
    {
```

```
        return -1; // Error: invalid range
    }

    // Generating random num
    return int_min + rand() % (int_max - int_min + 1);
}

// Function to initialize a random population
void generate_population(int POPULATION_SIZE, int NUM_VARIABLES,
   double population[POPULATION_SIZE][NUM_VARIABLES], double Lbound
   [NUM_VARIABLES], double Ubound[NUM_VARIABLES])
{
    // Your CODE: randomly initialize for all values in "population
       [i][j]""
    for (int i = 0; i < POPULATION_SIZE; i++)
    {
        for (int j = 0; j < NUM_VARIABLES; j++)
        {
            population[i][j] = generate_random(Lbound[j], Ubound[j
               ]);
        }
    }
}

// Function to compute the objective function for each member of
   the population
void compute_objective_function(int POPULATION_SIZE, int
   NUM_VARIABLES, double population[POPULATION_SIZE][NUM_VARIABLES
   ], double fitness[POPULATION_SIZE])
{

    /* Your CODE: compute "fitness[i]"" for each set of decision
       variables (individual) or each row in "population"
    by calling "Objective_function" */
    for (int i = 0; i < POPULATION_SIZE; i++)
    {
        fitness[i] = Objective_function(NUM_VARIABLES, population[i
           ]);
    }
```

```
}

void crossover(int POPULATION_SIZE, int NUM_VARIABLES, double
   fitness[POPULATION_SIZE], double new_population[POPULATION_SIZE
   ][NUM_VARIABLES], double population[POPULATION_SIZE][
   NUM_VARIABLES], double crossover_rate)
{
    /* Your CODE: Implement the logic of crossover function here
       based on "fitness_probs" or each set
    of decision variables (individual) or each row in "population".
    And save the new population in "new_population"*/


// Calculate and normalize fitness probabilities
double fitness_probs[POPULATION_SIZE];
double sum_fitness = 0.0;

// Computing the inverse of fitness for probabilities and their sum
for (int i = 0; i < POPULATION_SIZE; i++) {
    fitness_probs[i] = 1.0 / fitness[i];
    sum_fitness += fitness_probs[i];
}

// Normalizing the probabilities
for (int i = 0; i < POPULATION_SIZE; i++) {
    fitness_probs[i] /= sum_fitness;
}

// Calculating cumulative probabilities
double cumulative_probs[POPULATION_SIZE];
cumulative_probs[0] = fitness_probs[0];
for (int i = 1; i < POPULATION_SIZE; i++) {
    cumulative_probs[i] = cumulative_probs[i - 1] + fitness_probs[i
       ];
}

// Selecting parents based on cumulative probs
for (int i = 0; i < POPULATION_SIZE; i += 2) {
    int parent1 = -1;
```

```
    int parent2 = -1;
    double rand_num1 = generate_random(0.0, 1.0);
    double rand_num2 = generate_random(0.0, 1.0);

    // Selecting parent1
    for (int j = 0; j < POPULATION_SIZE; j++) {
        if (rand_num1 <= cumulative_probs[j]) {
            parent1 = j;
            break;
        }
    }

    // Selecting parent2
    for (int j = 0; j < POPULATION_SIZE; j++) {
        if (rand_num2 <= cumulative_probs[j]) {
            parent2 = j;
            break;
        }
    }

    // Performing crossover based on crossover rate
    if (generate_random(0.0, 1.0) < crossover_rate) {
        int crossover_index = rand() % NUM_VARIABLES;

        // Crossover genetics
        for (int j = 0; j < NUM_VARIABLES; j++) {
            if (j < crossover_index) {
                new_population[i][j] = population[parent1][j];
                new_population[i + 1][j] = population[parent2][j];
            } else {
                new_population[i][j] = population[parent2][j];
                new_population[i + 1][j] = population[parent1][j];
            }
        }
    } else {
        // Copying parents to new population if no crossover
           happens
        for (int j = 0; j < NUM_VARIABLES; j++) {
            new_population[i][j] = population[parent1][j];
```

```
            new_population[i + 1][j] = population[parent2][j];
        }
    }
}

// THIS WAS MY OLD CODE, I LEFT IT HERE SO I CAN COMPARE WITH THE
   NEWER ONE ABOVE, I ADDED FITNESS PROBABILITIES SO ITS MORE
   EFFICIENT AND RANDOMIZED

    // int parent_indices[POPULATION_SIZE]; // creat array to keep
       track of parents selected
    // for (int i = 0; i < POPULATION_SIZE; i++)
    // {
    //     parent_indices[i] = i; // initializes indices
    // }

    // // performing crossover in pairs of individuals
    // for (int i = 0; i < POPULATION_SIZE; i += 2)
    // {
    //     if (i + 1 > POPULATION_SIZE)
    //         break; // enrue pair to crossover

    //     // check if crossover should happen
    //     if ((double)rand() / RAND_MAX < crossover_rate)
    //     {
    //         int crossover_point = rand() % NUM_VARIABLES;

    //         for (int j = 0; j < NUM_VARIABLES; j++)
    //         {
    //             if (j < crossover_point)
    //             {
    //                 new_population[i][j] = population[i][j];
    //                 new_population[i + 1][j] = population[i +
       1][j];
    //             }
    //             else
    //             {
    //                 new_population[i][j] = population[i + 1][j];
    //                 new_population[i + 1][j] = population[i][j];
```

```
    //              }
    //          }
    //      }
    //      else
    //      {
    //          // no crossover, just copying parents to new pop
    //          for (int j = 0; j < NUM_VARIABLES; j++)
    //          {
    //              new_population[i][j] = population[i][j];
    //              new_population[i][j] = population[i + 1][j];
    //          }
    //      }
    // }
}

void mutate(int POPULATION_SIZE, int NUM_VARIABLES, double
   new_population[POPULATION_SIZE][NUM_VARIABLES], double
   population[POPULATION_SIZE][NUM_VARIABLES], double Lbound[
   NUM_VARIABLES], double Ubound[NUM_VARIABLES], double mutate_rate
   )
{
    /*Your CODE: Implement the logic of mutation on "new_population
       " and then copy everything into "population"*/

    // Loop through each individual in the new population
    // for (int i = 0; i < POPULATION_SIZE; i++)
    // {
    //      // Loop through each variable for the individual
    //      for (int j = 0; j < NUM_VARIABLES; j++)
    //      {
    //          // Check if mutation should occur based on the
       mutation rate
    //          if ((double)rand() / RAND_MAX < mutate_rate)
    //          {
    //              // Set the variable to a new random value
    //              new_population[i][j] = generate_random(Lbound[j
       ], Ubound[j]);
    //          }
    //      }
```

```
    // }
    for (int i = 0; i < POPULATION_SIZE; i++)
    {
        for (int j = 0; j < NUM_VARIABLES; j++)
        {
            if ((double)rand() / RAND_MAX < mutate_rate)
            {
                new_population[i][j] = generate_random(Lbound[j],
                    Ubound[j]);

            }
        }
    }

    // Copy the mutated individuals back to the old population
    for (int i = 0; i < POPULATION_SIZE; i++)
    {
        for (int j = 0; j < NUM_VARIABLES; j++)
        {
            population[i][j] = new_population[i][j];
        }
    }
}
```

GA.c

```
// Including everything necessary here
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "functions.h"
//#include "OF.h"

#ifndef M_PI
#define M_PI 3.14159265358979323846264338327950288419 7
#endif

int main(int argc, char *argv[])
{
```

```
// ###############################################################
   #######################
// YOUR CODE: Handle the possible errors in input data given by
    the user and say how to execute the code

// YOUR CODE: Assign all inputs given by the user argv[i] like:
// POPULATION_SIZE, MAX_GENERATIONS, crossover_rate,
   mutate_rate, stop_criteria

if (argc != 6)
{
    printf("Usage: %s <Pop Size> <Max Generations> <Crossover
       Rate> <Mutation Rate> <Stop Criteria>\n", argv[0]);
    printf("Please use proper amount of arguments \n");
    return 1;
}


int Pop_size = atoi(argv[1]);
int Max_gen = atoi(argv[2]);
double Cross_rate = atof(argv[3]);
double Mut_rate = atof(argv[4]);
double Stop_crit = atof(argv[5]);

// ###############################################################
   #######################
/* YOUR CODE: You must change this part based on the lower and
   upper bounds
1. based on what function is going to be minimized (let's say
   Levy)
2. given bound in https://www.sfu.ca/~ssurjano/optimization.
   html for each function.*/

// the number of variables (d)

srand(time(NULL)); // to make different num for rand each time
```

```
int NUM_VARIABLES = 10;
// the lower bounds of variables (x_1, x_2, ..., x_d) where d=
   NUM_VARIABLES
double Lbound[] = {-5, -5, -5, -5, -5, -5, -5, -5, -5, -5};
// the upper bounds of variable
double Ubound[] = {+5, +5, +5, +5, +5, +5, +5, +5, +5, +5 };

/*For example: in Levy function x_i     [-10, 10], for all i =
   1,    , d. This means:
lower bound = -10 for all x_i
upper bound = +10 for all x_i
if d =10 (or NUM_VARIABLES = 10) then:
double Lbound[] = {-5.0, -5.0, -5.0, -5.0, -5.0, -5.0, -5.0,
   -5.0, -5.0, -5.0};
double Ubound[] = {+5.0, +5.0, +5.0, +5.0, +5.0, +5.0, +5.0,
   +5.0, +5.0, +5.0};

Another example: in Griewank() if NUM_VARIABLES = 7, then:
// double Lbound[] = {-600.0, -600.0, -600.0, -600.0, -600.0,
   -600.0, -600.0};
// double Ubound[] = {+600.0, +600.0, +600.0, +600.0, +600.0,
   +600.0, +600.0};
*/

// ############################################################
   #######################

// YOUR CODE: Here make all the initial print outs

printf("Genetic Algorithm Parameters:\n");
printf("Population Size: %d\n", Pop_size);
printf("Max Generations: %d\n", Max_gen);
printf("Crossover Rate: %.2f\n", Cross_rate);
printf("Mutation Rate: %.2f\n", Mut_rate);
printf("Stop Criteria: %.2f\n", Stop_crit);

clock_t start_time, end_time;
double cpu_time_used;
start_time = clock();
```

```
// <YOUR CODE: Declare all the arrays you need here>

double population[Pop_size][NUM_VARIABLES];
double new_population[Pop_size][NUM_VARIABLES];
double fitness[Pop_size];
double best_solution[NUM_VARIABLES]; //
double best_fitness = INFINITY; //
double best_fitness_bef;
int gen_counter = 0;
int maxgen_count = 3000; // change as you go


// <YOUR CODE: Call generate_population function to initialize
   the "population"> like:
// generate_population(POPULATION_SIZE, NUM_VARIABLES,
   population, Lbound, Ubound);
generate_population(Pop_size, NUM_VARIABLES, population, Lbound
   , Ubound);

// iteration starts here. The loop continues until
   MAX_GENERATIONS is reached
// Or stopping criteria is met
for (int generation = 0; generation < Max_gen; generation++)
{
    // <YOUR CODE: Compute the fitness values using objective
       function for
    // each row in "population" (each set of variables)> like:
    compute_objective_function(Pop_size, NUM_VARIABLES,
       population, fitness);

    // <YOUR CODE: Here implement the logic of finding best
       solution with minimum fitness value
    // and the stopping criteria>

    for (int i = 0; i < Pop_size; i++)
    {
        if (fitness[i] < best_fitness)
        {
```

```
                best_fitness_bef = best_fitness;
                best_fitness = fitness[i];
                for (int j = 0; j < NUM_VARIABLES; j++)
                {
                    best_solution[j] = population[i][j];
                }
                gen_counter = 0;
            }
        }

        gen_counter+=1;

        // Check stopping criteria
        if (fabs(best_fitness - best_fitness_bef) < Stop_crit)
        {
            printf("Stopping criteria met at generation %d.\n",
               generation);
            break;
        }

        if (gen_counter >= maxgen_count)
        {
            break;
        }


        // <YOUR CODE: Here call the crossover function>
        crossover(Pop_size, NUM_VARIABLES, fitness, new_population,
            population, Cross_rate);
        // <YOUR CODE: Here call the mutation function>
        mutate(Pop_size, NUM_VARIABLES, new_population, population,
            Lbound, Ubound, Mut_rate);
        // Now you have the a new population, and it goes to the
           beginning of loop to re-compute all again
    }

    // ############################################################
       #######################
    // You dont need to change anything here
```

```
    // Here we print the CPU time taken for your code
    end_time = clock();
    cpu_time_used = ((double)(end_time - start_time)) /
       CLOCKS_PER_SEC;
    printf("CPU time: %f seconds\n", cpu_time_used);
    // ##############################################################
       #######################

    // <Your CODE: Here print out the best solution and objective
       function value for the best solution like the format>

    printf("Best solution found:\n");
    for (int j = 0; j < NUM_VARIABLES; j++)
    {
        printf("x[%d] = %f\n", j, best_solution[j]);
    }
    printf("Objective function value: %f\n", best_fitness);
    printf("\n");
    printf("Best fitness: %e\n", best_fitness);

    return 0;
}
```

Makefile

```
 # Variables for compiler and flags
CC = gcc
CFLAGS = -Wall -O2
LM = -lm

# Targets and their dependencies
all: GA

# Building the final executable
GA: functions.o OF.o GA.o
 $(CC) $(CFLAGS) -o GA functions.o OF.o GA.o $(LM)

# Compile functions.c
functions.o: functions.c
 $(CC) $(CFLAGS) -c functions.c -o functions.o
```

```
# Compile OF.c
OF.o: OF.c
	$(CC) $(CFLAGS) -c OF.c -o OF.o

# Compile GA.c
GA.o: GA.c
	$(CC) $(CFLAGS) -c GA.c -o GA.o

# Clean rule to remove object files and executable
clean:
	rm -f *.o GA
```