MECHTRON 2MP3 - Porgramming for Mechatronics

Developing a Basic Genetic Optimization Algorithm in C

Marco Tan, 400433483

1 Introduction

2 Programming Genetic Algorithm

2.1 Implementation (12 points)

Files you have downloaded are:

- `OF.c` (Do not change anything in it)
- `functions.h` (Do not change anything in it)
- `GA.c`
- `functions.c`

```c
#include <math.h>
#include "functions.h"

double Objective_function(int NUM_VARIABLES, double x[
   NUM_VARIABLES])
{
 double sum1 = 0.0, sum2 = 0.0;
 for (int i = 0; i < NUM_VARIABLES; i++)
 {
  sum1 += x[i] * x[i];
  sum2 += cos(2.0 * M_PI * x[i]);
 }
 return -20.0 * exp(-0.2 * sqrt(sum1 / NUM_VARIABLES)) - exp(
    sum2 / NUM_VARIABLES) + 20.0 + M_E;
}
```

The number of decision variables in the optimization problem is set to 2, with upper and lower bounds for both decision variables set to +5 and -5, respectively:

```
int NUM_VARIABLES = 2;
double Lbound[] = {-5.0, -5.0};
double Ubound[] = {5.0, 5.0};
```

```
Genetic Algorithm is initiated.
------------------------------------------------
The number of variables: 2
Lower bounds: [-5.000000, -5.000000]
Upper bounds: [5.000000, 5.000000]

Population Size:   100
Max Generations:   10000
Crossover Rate:    0.500000
Mutation Rate:     0.100000
Stopping criteria: 0.0000000000000001

Results
------------------------------------------------
CPU time: 1.143935 seconds
Best solution found: (-0.0001323239496775, -0.0000225231135396)
Best fitness: 0.0003801313729501
```

Table 1: Results with Crossover Rate = 0.5 and Mutation Rate = 0.05, Stop Parameter = 0

Pop Size	Max Gen	Best Solution			CPU time (Sec)
		x_1	x_2	Fitness	
10	100	0.1128262677755334	0.0167843350287455	1.574235	0.0001200
100	100	0.0157086155450479	-0.0018225633547750	10.110173	0.0012430
1000	100	0.0021972204568783	0.0010652071801323	141.517785	0.0156620
10000	100	-0.0000612600706802	-0.0000696838833711	3792.785006	0.1752030
1000	1000	-0.0001671514474637	-0.0001416751184227	1030.382844	0.1493740
1000	10000	-0.0000269873999184	-0.0000024517066786	12875.682388	1.5293810
1000	100000	-0.0000051525421458	0.0000009662471712	53390.879056	14.9062560
1000	1000000	0.0000001234002411	0.0000001466833055	648436.305077	149.0835510

Table 2: Results with Crossover Rate = 0.5 and Mutation Rate = 0.05, Stop Parameter = 1e-16

Pop Size	Max Gen	Best Solution			CPU time (Sec)	Gens
		x_1	x_2	Fitness		
10	100	0.0337652373284865	0.2125752369000464	0.672888	0.0000770	64
100	100	0.0300162076158532	0.0395693560315156	4.876065	0.0001170	9
1000	100	-0.0178259168834547	0.0017210398808682	16.899683	0.0026420	20
10000	100	-0.0018487754286491	0.0018150242985291	133.197654	0.0270060	17
1000	1000	-0.0003592879513086	-0.0000489712693028	970.763295	0.0200700	155
1000	10000	0.0000723614357749	0.0000571995042531	3815.094110	0.4947670	3761
1000	100000	0.0000153691507947	-0.0000033271498996	21985.724392	2.2993190	17615
1000	1000000	0.0000002072192729	0.0000007334165275	316892.930285	13.9982390	106598

With the way I implemented the genetic algorithm, we only stop running the GA if we've reached the stop condition for a consecutive number of generations. The way I calculated how many generations we have to consecutively reach the stop condition is `MAX_GENERATIONS * 0.05`. As we can see, we get significantly better fitness if we run without a stop parameter versus when we do run with a stop parameter. This is because when we run the code without a stop parameter, we guarantee that the algorithm will run for the maximum number of generations allowed, which will converge our values closer to the global minimum. However, with a stop parameter, it is not guaranteed that our algorithm will run for that long, and so we get less fit values overall.

I also implemented elitism in my GA to see if it would impact performance in terms of execution speed and our overall fitness. Elitism is a strategy in GAs in which we allocate a set amount of space for the most fit individuals in a population. We allow the fittest individuals a guaranteed spot in the next generation, which should theoretically increase the population's overall fitness. In theory, this should help us converge towards the optimized value. Here's a table with elitism enabled in the code:

Table 3: Results with Crossover Rate = 0.5 and Mutation Rate = 0.05, Stop Parameter = 0, With Elitism, Elitism Rate = 0.1

Pop Size	Max Gen	Best Solution			CPU time (Sec)
		x_1	x_2	Fitness	
10	100	-0.5699467731499794	-0.0808229227926685	0.299646	0.0007130
100	100	0.0166529440398575	-0.0265371683177245	8.727955	0.0487070
1000	100	0.0073487148654401	-0.0035749678516641	40.172230	4.4630050
10000	100	-0.0002649822273595	0.0002302718349876	1002.788198	369.5191410
1000	1000	0.0002504023724468	-0.0000487011857562	1380.741144	38.1626000
1000	10000	-0.0000081094913220	0.0000106659717902	25705.486871	388.3493910

Table 4: Results with Crossover Rate = 0.5 and Mutation Rate = 0.05, Stop Parameter = 1e-16, With Elitism, Elitism Rate = 0.1

Pop Size	Max Gen	Best Solution			CPU time (Sec)	Gens
		x_1	x_2	Fitness		
10	100	0.2661389276693296	0.2415573062568699	0.365861	0.0003410	48
100	100	-0.0376040325675184	-0.1676733396796850	0.929195	0.0047660	12
1000	100	0.0048563512996100	0.0050673889951156	47.249322	0.7756500	21
10000	100	0.0009297556248171	-0.0024037784907982	133.911033	75.8202890	21
1000	1000	0.0030676205656812	0.0051449728222313	55.869929	2.0167950	54
1000	10000	-0.0000632251613135	-0.0000023585744211	5553.743165	58.5307230	1534

(This would be a lot faster if I made this with parallelism in OpenMP in mind, but for now, this is fine.)

Here, I compiled my code with the `-D ELITISM` compile flag to tell the compile to define `ELITISM`. I have a bunch of preprocessor directives in my code that check this definition and will include sections for implementing elitism when it is defined.

As we can see, elitism doesn't necessarily make the results better since it's still dependent on the randomness of our initial population. This means that there's still a chance that we start our population off with a local minimum and because elitism guarantees the most fit individuals continue onward, we propagate that local minimum further. We only get out of that local minimum once mutation introduces enough genetic diversity to get us out. We also see that the time complexity has gone significantly up. I'm sure I could've optimized this further to make it faster (like using a parallel merge sort instead of C's `qsort`), but this is fine for now.

There were other strategies that I wanted to explore as well, such as repopulation (where we kill lower-fitness individuals in the population and replace them with new, randomly generated individuals) or the idea of using a circular genome for individuals in our population, but for now, this is sufficient. I'm hoping that maybe later, I could write my genetic algorithm (without template files) to further explore these concepts.

2.2 Report and `Makefile` (3 points)

The following is the contents of my `Makefile`.

```
CC=gcc
CFLAGS=-lm -Ofast
DFLAGS=-Wall -Wextra -W -pedantic -O0 -g -pg -lm
ELITISM=

DEPS = functions.h
OBJS = GA.o OF.o functions.o

%.o: %.c $(DEPS)
    $(CC) -c -o $@ $< $(CFLAGS)

GA: $(OBJS)
    $(CC) -o $@ $^ $(CFLAGS)

.PHONY: clean

clean:
    rm -f *.o GA
```

- `CC` defines the compiler we're using.
- `CFLAGS` is all our compile flags.
  - `-lm` includes `<math.h>`
  - `-Ofast` tells GCC to optimize the code as much as possible, for a little performance boost.
- `DFLAGS` are the compile flags I use when I'm still debugging the code.
  - `-Wall, -Wextra, -W` are all warning flags to enable warnings that may be important to me during compile time.
  - `-pedantic` ensures that I'm adhering to ANSI C. It doesn't matter much, but it's nice to have.
  - `-O0` ensures that the compiler does not optimize anything, such that the output when we debug is exactly what we expect.
  - `-g` is the flag we use if we want the binary to be compatible with `gdb`.

  - `-pg` is the flag we use if we want the binary to be compatible with `gprog`.
  - `-lm` is same as above, including `<math.h>`.

- `ELITISM` can be filled with a `-D ELITISM` flag if you want to compile the program with elitism.

Our `DEPS` variable lists all header file dependencies that this program requires. `OBJS` is a variable listing all the object files we need to compile before we can link them to the main program.

The first declaration will compile all source code files in that directory, into object files (`*.o`). The second declaration will gather the object files together and link them into the final output executable, `GA`.

`.PHONY` states that the following declaration isn't to compile files, but as a helpful script we could run. I defined a `clean` to clean the directory of the binary and object files if I wish to rebuild the code completely.

2.3 Improving the Performance - Bonus (+1 points)

Using `gprof`, I realized that the most time-consuming task was the crossover function. My old implementation used a `for` loop, iterating through a list of fitness and summing them successively so I could pick parents based on a roulette wheel selection algorithm. However, that would be $\mathcal{O}(n)$ time complexity and since I have to do it twice (because I have to pick two parents), it makes the code significantly slower.

Instead of this, I sum up the probabilities beforehand and use a binary search to pick parents during the crossover. Binary search essentially divides a sorted list into two over and over again, narrowing the search space that contains the value we want until we get to the value. Binary search has a time complexity of $\mathcal{O}(\log n)$, which is significantly better performance than the iterative `for` loop.

Table 5: Results with Crossover Rate = 0.5 and Mutation Rate = 0.2, Stop Parameter = 0, No Elitism

Pop Size	Max Gen	Best Solution			CPU time (Sec)
		x_1	x_2	Fitness	
10	100	0.0232250546213351	0.0016808952212708	12.252658	0.0001190
100	100	0.0062705506599841	-0.0028653233325411	48.155814	0.0012490
1000	100	0.0004579941744254	-0.0000161421485325	647.887689	0.0150990
10000	100	0.0001305690967159	0.0000147777609598	2680.083076	0.1791700
1000	1000	0.0000298605300628	-0.0000156904570829	10369.297786	0.1543050
1000	10000	0.0000148243270885	-0.0000003376044333	23284.917118	1.4530580
1000	100000	-0.0000007892958829	-0.0000000116415322	308228.189286	14.5638270
1000	1000000	-0.0000000116415322	-0.0000000349245965	707373.784597	145.2967410

Table 6: Results with Crossover Rate = 0.5 and Mutation Rate = 0.2, Stop Parameter = 1e-16, No Elitism

Pop Size	Max Gen	Best Solution			CPU time (Sec)	Gens
		x_1	x_2	Fitness		
10	100	-0.8474522437189949	0.1049383194674451	0.331737	0.0000220	7
100	100	-0.0149111985298394	-0.0155445956697431	13.653199	0.0005510	42
1000	100	-0.0012676022021409	-0.0009034783583619	223.797324	0.0022020	15
10000	100	0.0004406436348523	0.0003503332847501	624.348251	0.0243050	14
1000	1000	0.0000673555769337	-0.0001193000935569	2570.712165	0.0768130	530
1000	10000	-0.0000805058516935	0.0000638631172780	3425.462826	0.1676250	1144
1000	100000	0.0000046216882783	-0.0000070151872968	40383.104333	2.1756900	14654
1000	1000000	-0.0000001885928214	-0.0000007566995928	311938.809237	33.3154300	228397