MECHTRON 2MP3 Assignment 2

Developing a Basic Genetic Optimization Algorithm in C

Andrew De Rango, 400455362

1 Programming the Genetic Algorithm

1.1 Implementation

The code can be broken down into 4 files:

- `OF.c`: Computes the Ackley function given any number of variables.
- `functions.h`: Contains function prototypes.
- `GA.c`: Contains the main function.
- `functions.c`: Defines and executes functions such as crossover, mutation, population initialization, etc.

The number of decision variables in the optimization problem is set to 2 by default, with upper and lower bounds for both decision variables set to +5 and -5, respectively. This is the recommended search range for a two-dimensional optimization test for the Ackley function. To change this, visit the following lines:

```
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:     0.200000
```

```
Stopping criteria: 0.0000000000000001

Results
-----------------------------------------------
CPU time: 0.012754 seconds
Best solution found: (-0.000127286184639, 0.002088050824631)
Best fitness: 0.006033388506971
```

Table 1: Results with Crossover Rate = 0.5 and Mutation Rate = 0.05

Pop Size	Max Gen	Best Solution			CPU time (Sec)
		x_1	x_2	Fitness	
10	100	0.043437976876012	-0.904300755776605	2.677441414918161	0.000115
100	100	0.004633187784177	0.004663432484802	0.019743731092440	0.001359
1000	100	-0.000112175475858	0.000729511957955	0.002102128938884	0.009205
10000	100	0.000066997017742	0.000116287265027	0.000380072324405	0.377696
1000	1000	0.002301421483187	0.000252665486304	0.006691246774786	0.009532
1000	10000	-0.000127286184639	0.002088050824631	0.006033388506971	0.012754
1000	100000	-0.004743256142709	0.009201655634307	0.032132114007670	0.012977
1000	1000000	0.009314462081210	-0.004746329507207	0.032476328127761	0.017277

Table 2: Results with Crossover Rate = 0.5 and Mutation Rate = 0.2

Pop Size	Max Gen	Best Solution			CPU time (Sec)
		x_1	x_2	Fitness	
10	100	-0.011278062598444	0.019696436365925	0.077864075069045	0.000080
100	100	0.043074449078681	0.020379226198597	0.194258202498812	0.001218
1000	100	-0.000641841907353	-0.000256060157091	0.001967254744301	0.037911
10000	100	0.000013338867582	0.000035285484062	0.000106733375379	0.910979
1000	1000	0.001356226858383	0.002428398003070	0.008073129104510	0.028995
1000	10000	0.000922891777438	-0.001703875605810	0.005580803229666	0.029390
1000	100000	-0.001054171007617	0.000625157263421	0.003506521480777	0.025726
1000	1000000	-0.000289760064469	-0.000070461537722	0.000845816690682	0.043721

Note that the above tables were produced without the implementation of the optimized crossover and mutation function. The optimized crossover and mutation functions provide significant increases to the genetic algorithm's accuracy. The results are shown below and can be compared to the tables above to see the effect of the improved crossover and mutation functions.

1.2 Makefile Summary

The Makefile supplied with the Genetic Algorithm repository can be seen below:

```
CC = gcc
            CFLAGS = -Wall -g -O3
            TARGET = GA
            SRC = GA.c functions.c OF.c
            OBJ = $(SRC:.c=.o)

            $(TARGET): $(OBJ)
             $(CC) $(CFLAGS) -o $(TARGET) $(OBJ)

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

            clean:
             rm -f $(OBJ) $(TARGET)
```

This Makefile is a set of instructions instigated from the command line that aids in the compilation and linking processes of potentially multiple source code files. An example is shown in the code block above. Here are the roles of the individual components within the Makefile:

`CC=gcc`: CC sets the compiler that will be used to compile the program. In this case, we are using gcc.

`CFLAGS= -Wall -g -O3`: CFLAGS lists the flags that will be used by the compiler defined above. Each dash represents a precursor for another flag. -Wall enables the compiler to display warning messages upon compilation, such as declared but unused variables within the program. -g enables the compiler to provide debugging information, and -O3 enables possible optimizations within the program to further reduce the computation time.

`TARGET=GA`: TARGET defines the name of the final executable that will be placed in the same directory. This name ('GA' in this case) will need to be called in the command line to run the program.

`SRC=GA.c functions.c OF.c`: Define the source files containing C code that need to be compiled. Did not include functions.h because it is a header file and only contains function prototypes; not executable C code.

`OBJ=$(SRC:.c=.o)`: Creates object files with .o extension for each of the files defined in the SRC component above.

`$(TARGET): $(OBJ)`: This states that the target files are dependent on the object files. Thus, if there are changes made to the object files defined above, the target (GA) will need to be rebuilt.

`$(CC) $(CFLAGS) -o $(TARGET) $(OBJ)`: Responsible for building the target file, GA, from the compiler. With this, it sends along the flags and other specified parameters as defined above.

`%.o: %.c`: This is a pattern rule. It defines how to generate the .o file from the .c file.

`$(CC) $(CFLAGS) -c $<`: Command that is run which helps the construction of an object file from a C file. This command passes the flags specified in `CFLAGS`. The -c option causes the compiler to generate the object file.

`clean`: This is an independent rule. It does not depend on any of the parameters defined above.

`rm -f $(OBJ) $(TARGET)`: If the user runs the command `make clean` in the command line, then this is the function that will be executed. It removes the object files and final executable to clean up to the repository.

1.3 Improving the Performance - Bonus

A variety of changes were made to the `functions.c` file. Some were made to hasten the computational process, and some to improve the accuracy of the program. Permission was given by the instructor to alter the `functions.h` in order to pass the generations variable into the mutation function, on the basis of significant accuracy improvement. The pronounced changes are summarized below. The altered `functions.h` file has been submitted.

The first and most important change was to the mutation function. Instead of randomly mutating to any point within the defined range, which provides a very small probability that the mutation will drive a strong solution. Therefore, I decided to adjust the mutation to be generation-dependent. As the generations increase, the range of which genes mutate converges to 0 about the point of best fitness. This works with functions whose global minimum isn't (0, 0, ...) because the early generations are likely to find the hole of the global max, and then the range slowly converges around that point to permit extreme accuracy. So for the first generations, the mutation range is still as defined or very close, which permits genetic diversity. This technique was validated on a variety of functions with large ranges, more independent variables, and different characteristics. This includes the Ackley function, the Rastrigin function, the Beale function, the Levi function N. 13, and the Himmelblau function. The results were all very close to the true global minimum.

Next, I changed the method in which certain individuals were selected, based on any probability regardless of how the probability was calculated. This was done by creating another array that summed each of the probabilities, and then generated a random double between 0 and 1, then found the individual that corresponded to this random double. The optimized part was that this search was done using binary search, which improved the time complexity to logarithmic. Previously,

this was done by starting from the first index and iterating through the array until the associated individual was found. This change produced a noticeable effect.

Finally, I also changed the crossover function so that more fit individuals are more likely to have their genes crossed than less fit individuals. The probabilities were based on a normalized value of the exponential of their z-score relative to the rest of the population. This provides extra crossover strength to very strong individuals, and removes crossover strength from weaker individuals, even more so relative to the simple normalized 1/fitness used for determining intergenerational individuals.

Although these changes decrease the genetic diversity of the population, they do so at a sufficiently late generation, so that the global minimum is found prior to convergence. The technique may render bad results with very low population counts or a very large initial search range, but these are functions that also pose difficulties to standard genetic algorithms.

These adjustments provide the following results:

Table 3: Results with Crossover Rate = 0.5 and Mutation Rate = 0.2

Pop Size	Max Gen	Best Solution			CPU time (Sec)
		x_1	x_2	Fitness	
100	100	0.000000064633813	-0.000000348628304	0.000001002876119	0.004836
1000	100	0.000000000000000	-0.000000000000000	0.000000000000000	0.023476
10000	100	0.000000000000000	-0.000000000000000	0.000000000000000	0.285449
1000	1000	0.000000000000000	0.000000000000000	0.000000000000000	0.022813
1000	10000	0.000000000000000	0.000000000000000	0.000000000000000	0.022514
1000	100000	-0.000000000000000	0.000000000000000	0.000000000000000	0.023436