CP468 Artificial Intelligence

Simple Genetic Algorithm

11 December 2022

Adam Scott 190600780 scot0780@mylaurier.ca

Tatiana Olenciuc 191001870 olen1870@mylaurier.ca

Saitan Taneja 200744020 tane4020@mylaurier.ca

Alex Lau 190786790 laux6790@mylaurier.ca

Pranav Patel 200698660 pate9866@mylauirer.ca

Discussion

The Simple Genetic Algorithm (SGA) is an Artificial Intelligence technique that can find the inputs to a black-box objective function (OF) such that this function is maximized/minimized.

We used Python strings to represent the binary vectors. These were used because string manipulation in Python is very straightforward, making programming and testing easier, as well as creating more readable code.. For the three classical benchmark OFs, our implementation was to take a binary vector of 0s and 1s and convert it into a Cartesian coordinate between -5 and 5. We accomplished this by taking binary vectors as having size multiples of 16. If you have 32 bits, it's two dimensions of input. Each 16-bit block is decomposed into a short integer. The first bit is the sign bit and the remaining 15 give a number between -32767 and 32767. This number is then divided by 32767 and multiplied by 5 to give us 65535 possible numbers between -5 and 5.

For the additional OFs provided, no translation into Cartesian coordinates was necessary since those OFs compute their results directly from the bits.

To implement the biased roulette, we first sorted the list, then performed a weighted coin flip to determine whether a vector makes it to the tentative next generation. The best vector has a 25% chance of being selected. If it isn't, then, the next vector has a 25% chance and so on until one is selected. We repeat this process until we have the same number of tentative new generation members as we started with. If the population size were 100, on average, the best vector would be 1 * 25/100 = 25% of the new population, the next would make up 0.75 * (25/100) = 18.75%, and so on. This makes it so that individuals with lower OF values pass on their genes more often.

The loop of reproduction, crossover, and mutation eventually results in a binary vector that minimizes the OF. However, we did run into some challenges which we will outline below.

The de Jong function decreases towards the global maximum in the same way from all directions. This means that it is fairly simple for the binary vectors to find their way towards the global minimum. However, this is more tricky for the Rosenbrock and Himmelblau functions. The binary vectors may get stuck in a rut far away from the global minima. To combat this, we check each generation whether the last generation's fittest individual is within 0.001 of this generation's. If so, we generate a new random population and restart the process.

The second major hurdle of this implementation is that the inputs may not give an exactly 0 output. For Rosenbrock and Himmelblau, there isn't a binary encoding that gives exactly 0 as an output to the OF. Our solution to this is to accept any values 0.00xxxx… as solutions. If the OF gives less than a hundredth as the objective value, we say that we're about the global minimum and stop looking.

Installation and Code Execution

This program is written in Python and has been tested on Windows 10 machines running Python 3.10. Older versions of Python and other operating systems should work alright.

1. 1. Create sga.py on your computer, making sure it contains all of the code below in this document. There is only one file, sga.py. Administrator privileges should not be necessary for the code to execute. Here is a Github link to download it from, so that you don't have to copy and paste it from the text.
2. Two options for execution:

   a. Open sga.py in a code editor configured to use a Python interpreter and run the file.
   b. Open a terminal, navigate to the folder containing sga.py, and type: 'python sga.py'.

3. After execution, some text files will be created. There is one for each objective function which has information about the best individual in each generation, as well as summary.txt, which tells you about the very last generation of all OFs, i.e., the solutions that were found.

Example Output (Summary.txt)

DE JONG

Population Size = 16

Vector Length = 32

Fittest member of gen 11 is: 10000000000000000000000000000000 with objective function value of: 0.0

ROSENBROCK

Population Size = 16

Vector Length = 32

Fittest member of gen 137 is: 10011001011101101001100101100011 with objective function value of: 0.003529658965565565

Number of resets: 2

HIMMELBLAU

Population Size = 16

Vector Length = 32

Fittest member of gen 820 is: 11001100101011011011001101111010 with objective function value of: 0.0018191579029568795

Number of resets: 15

2CCOF.25.C

Population Size = 16

Vector Length = 51

Fittest member of gen 18 is: 000000000000000000000000100000000000000000001000000 with objective function value of: 24

2CCOF.29.C

Population Size = 16

Vector Length = 59

Fittest member of gen 21 is:

10000000000001000000000000000000000000000001000000000000000 with objective function value of: 28

2CCOF.99.C

Population Size = 16

Vector Length = 199

Fittest member of gen 2946 is:

000000000000000000000000000000000000000000000000000000001000000000000000000000

000000000000000000000000000000000000000000000000000000000000000000000000000000

0100000000000000000000000000000000000000000 with objective function value of: 98