Lab Session – Evolutionary Computing

Scope

Genetic Algorithms are powerful global optimization tools with some interesting features. As stochastic algorithms, they rely on random search processes, coupled with fitness-based selection. Here, we'll look at a Genetic Algorithm written in Python, which we'll use to solve a simple one-max problem.

The problem

One-max is a binary test problem, where solutions get better fitness scores as the number of 1s in their genome increases. Given a 4-bit one-max problem, the best solution is

1111

and the worst is

0000.

Here, we will optimize a 30 bit one-max.

Overview of the code

We optimize a population of *Individual* objects. Each has a *fitness* (measuring how good it is) and a *genome* (in this case a bitstring), plus various associated methods.

The algorithm operates over a number of generations. In each generation, it creates a new population by:

- Preserving the current best solution
- Creating new children from members of the current population, introducing variation through random mutations and crossover

New individuals are evaluated and scored according to how they perform on the one-max problem.

The code finishes when either a set number of generations have passed, or the optimal solution has been found. When it is finished, it prints a graph showing the fitness progression of the experiment through the generations.

Tasks

The code is currently missing some key functionality, which you will need to complete.

1. To get the code to run properly, you need to implement a crossover method. In one-point crossover, we select a point in the child's genome and overwrite everything after that point with the genome from another child. Note that the *other* child needs to be passed as an argument – for now let's select the other child using random selection.
2. To help you see what each individual looks like, implement the printIndividual method, to print out the genome and fitness to screen. You may find this useful later on when you are investigating how the algorithm is performing.
3. Run the code a few times and note the graphs you generate. What types of behavior do you observe? How does the code perform? How long does it take to find the optimum?
4. We currently use random selection to pick a child from the population. Tournament selection can give much better performance. Implement tournament selection following the guidelines in the code. Switch any calls to randomSelect() to an equivalent call to tournamentSelect(). Run the code and note the behavior. Change the tournament size to 2, 4, 8, and 16. Check the graphs that are produced – how do you explain the performance of the algorithm?
5. Genetic algorithms very on the setting of two rates, mu (MUTATION_RATE, which controls how much mutation happens), and chi (CROSSOVER_RATE, which controls how often crossover occurs). For all combinations of mu=0.01, 0.05, 0.1, 0.2, and chi=0.2, 0.5, 0.9, how does the algorithm perform? What is the best setting for this problem?
6. One-point crossover can be replaced with two-point crossover. Select an *other* using tournament selection. Randomly select two points in the child's genome, and between those points, substitute in the corresponding section of the *other's* genome. Run the code a few times using one point and two point crossover – what differences do you observe?
7. A 30 bit problem such as this can be solved quite easily. What happens when the problem is set to 300 bits? If you are having difficulty solving a 300 bit one-max, try changing the population size, mu, and chi. What are the best settings that you've found for the 300 bit one-max. What about 3000? You may have to change the REPORT_FREQUENCY.
8. Eliteism is the process of preserving (without change) some of the previous population into the next population. Try ELITISM_NUMBERs from 2 to 15 – what happens to the speed of search? What happens when ELITISM_NUMBER is 0?
9. Finally, we can check out a different selection method. Fitness-proportionate selection (sometimes called roulette wheel) assigns a

selection probability for an individual to become a child, proportionally large to its fitness. Implement roulette wheel selection and contrast to tournament. Be careful of integer division – make sure variables are treated as floats as needed. Switch any calls to tournamentSelect() to an equivalent call to fitnessPropSelect().

a. Sum all fitnesses to create a total fitness for the population
b. Create a random number up to the total fitness (this is like spinning the roulette wheel)
c. Starting from 0, re-sum the fitnesses until you find the individual that was selected by the random number. Return this individual.