Practical 1: Simple Genetic Algorithm

The aim of this session is to give you practical experience of coding a Genetic Algorithm (GA) and experimenting with the effect of changing the use of different parameters and operators.

You can either choose to use BlueJ or any other editor that you are comfortable with. Remember to **save your code on raptor**, since we will be using it in future practical sessions.

Basics

We will use the GA to solve the famous OneMax problem (or BitCounting). It is a simple problem consisting in maximizing the number of ones of a bitstring. Individuals are represented by fixed-length boolean arrays (0 = false, 1 = true). Download from Moodle the skeleton GA project (material under Week 2). Open it on BlueJ (if you are using a different editor, open the file GA.java).

Take your time to familiarise yourself with the code. Check how the population is represented (look for the attribute `population`). Also, take a look at the constants and make sure you understand what they represent. Ask your class supervisor if you are in doubt. You will see that the main loop of the GA is partially implemented as the method `run`.

While the project compiles, you will see that no evolution takes place. Create a new instance of the class GA and execute the method `run`. What happens?

Exercise 1

Go to the method `initialise` in your GA class. You will see that the implementation is empty. Your task is to implement the initialisation procedure of the GA.

Remember that individuals are initialised with random values. To generate random values in java, use the class `java.util.Random`:
http://docs.oracle.com/javase/7/docs/api/java/util/Random.html

You will find that there is already an attribute called `random` that can be used to generate random numbers. Find out the method that you need to use to generate random boolean values.

Make sure that your method initialises all individuals in the population.

Exercise 2

The fitness function for the OneMax problem is simply the sum of the bits active (number of 1's). Go to the method `evaluate` and complete the implementation

to calculate the fitness of all individuals. Make sure that your method uses the attribute `fitness` to store each fitness values.

Exercise 3

Create a method `crossover(int first, int second)`. As you can see the method takes 2 parameters:

1) parameter `first`: this is the index of the first parent in the population array;
2) parameter `second`: this is the index of the second parent in the population array;

Complete the implementation to perform a one-point crossover. Make sure **not to modify** the data from the population array. The crossover method should return a 2-dimensional `boolean[][]` array representing the 2 offspring generated.

Exercise 4

Create a method `mutation(int parent)`. As you can see the method takes 1 parameter:

1) parameter `parent`: this is the index of the parent in the population array;

Complete the implementation to perform a point mutation. Make sure **not to modify** the data from the population array. The mutation method should return a 1-dimensional `boolean[]` array representing the offspring generated.

Exercise 5

Complete the `run` method. The run method should create a new population and iteratively add new individuals to it. New individuals are added by selecting parents from the current population and subjecting them to genetic operators according to their probability.

Exercise 6

After completing the implementation, run the algorithm again. What result do you get now? What happens when you run the algorithm several times?

Few more things to try:

- increase the number of `BITS` of the individual. Run the GA again. Does it make any difference in relation to the best fitness at the end of the run? Can you elaborate a reason for the difference?
- change the probabilities of mutation and crossover. Check the difference in the results. Can you determine an optimal combination of probabilities?
- reduce the `POPULATION_SIZE` and `MAX_GENERATION` values. How these affect the results in term of fitness? Which one has the larger negative impact?

Take notes and discuss your observations with colleagues and the class supervisor.

Conclusion / Feedback

This week we've seen the basic structure of a GA and how we can implement some of the most basic and popular functions, such as population initialisation, fitness calculation, and operators implementation. Keep in mind that the problem we've seen today (OneMax) is a toy problem, which we used for demonstrating how a GA can solve such a task. In reality, when we normally use GAs, we use them for much more complicated problems and usually the length of the individuals tends to be much longer (not just 5 bits!) and/or we are not only going to be dealing with binary strings.

More about that in the following lectures and practicals!