EXERCISE GENETIC ALGORITHMS

BIOS13, Dept. of Biology, Lund, Sweden

by Anders Brodin, modified for R by Jonas Håkansson, John Waller and Jörgen Ripa

This exercise should give you a basic knowledge of how a genetic algorithm can be used as an optimization tool. The task that the algorithm solves is trivial, you can easily solve it in your head; find the maximum of and eight-digit binary number. This is of course 11111111 in base 2 or 255 in base 10. The idea is that the problem should be so simple so that you can concentrate on the code rather than the problem. In reality you should only create a GA to solve a problem that is too complex for analytical solutions.

The exercise is divided into three levels, i) basic, ii) advanced, and, iii) expert level. You should start with the basic level questions and then continue to the more advanced levels if you wish.

i) Basic level

Download the R-script GA.R from Canvas and save it in a suitable directory. The main function is called **GA**, and it has the following subfunctions:

Create_pop (creates a random population of solutions (=chromosomes) to start with)

Evaluate_Fitness (calculates the value of the binary numbers in the solutions)

Sort (sorts the solutions so that the best is on top and the worst at the end)

Reproduce (mates the best solutions so the new solutions (=offspring) are created)

Mutate (changes specific genes from 1 to 0 or from 0 to 1)

1. Run the code using `source`. (do it a few times to see the variation in the output)

Why are there two curves in the plot? What does the jagged curve represent?

2. a) How many solutions are there? (**pop_size**)

b) Each solution consists of a row of ones and zeros, how many?

c) What is the purpose of the variable **num_breed**?

d) What does the variable **pop** consist of?

e) The function **Evaluate_fitness** contains a for loop inside another for loop. What is the purpose of this construction?

f) In what way are the new solutions (=the offspring) inserted in the population of solutions?

g) Why is there a variable called **mutation_free** in the Mutate function? What would happen with the solutions without this variable (if it was set to zero?)?

3. Is this a binary or continuous GA? What are the advantages and disadvantages with these two types of GAs?

ii) Advanced level

Computers are deterministic machines and thus cannot produce truly random numbers. Most programming languages still include some way of producing numbers in a way that simulates randomness, in R one of these ways is the command **runif()** (which means random number from a uniform distribution). The number produced isn't truly random, but rather what's called a pseudo-random number, and it's random enough for our purposes. In RStudio you can access help either by selecting the subject in the script and pressing **F1**; or by writing "**help(topic)**" at the prompt. The "**help(topic)**" command works in standalone R too

4. Use help to explain how the command **runif()** works.

Look through all code again and make sure that you understand what all commands mean. If you understand everything you are now ready to try to create an own GA. The task is to find the minimum of the following function:

$$f(x,y) = x\sin(4x) + 1.1y\sin(2y), \quad 0 \le x \le 10, \qquad 0 \le y \le 10$$

5. Plot this function to get a feeling for what it looks like, use the following code:

```
f <- function(x,y) x*sin(4*x)+1.1*y*sin(2*y)

x <- seq(0,10, length=101)

y <- seq(0,10, length=101)

z <- outer(x,y,f) # calculate f(x,y) for each value of x and y

persp(x,y,z,theta=30, phi=30, expand=0.6, ticktype='detailed')
```

iii) Expert level.

Now change the code so that the GA finds the minimum of the sine function in the given range. In order to do this, you have to make it a continuous GA so that it can handle decimal numbers directly. To accomplish this, you have to change the following:

- You need two (rather than 8) genes in each solution, but these should be decimal numbers between 0 and 10 rather than 0 and 1. To do this change the code in the **Create_pop** function.

- You need a new **Reproduce** function. The best would be linear blending as explained in the lecture. Account for that this procedure produces three (not two) offspring.

- **Evaluate_Fitness** has to be changed completely. Instead of adding up a binary number it should put x and y values into the sine function.

- The **Mutate** function should produce new random numbers between 0 and 10.

- The task has changed from a maximization to a minimization. You have to reverse the Sort function so that is puts the solutions that produces the smallest values on top.

6. What is the minimum of the sine function with two decimals' precision?

7. What are the values of x and y at this point?

8. How many generations does your GA require to solve the problem? Give the average of ten runs.

iv) Professional programmer level (try this only if you think it will be fun)

9. The traveling salesman problem is easily solvable with a GA for up to ten destinations. It concerns a traveling salesman that need to visit a number of cities and, of course, would like to find the shortest distance by car between these places. Calculate the shortest distance by car between the following Swedish cities: Lund, Kalmar, Stockholm, Karlstad, Göteborg, Jönköping. In reality some roads are partially the same, but assume for simplicity that all roads are separate. The distances in km between the various cities are:

	Lund	Kalmar	Stockholm	Karlstad	Göteborg	Jönköping
Lund	0	267	602	506	262	282
Kalmar		0	412	467	341	214
Stockholm			0	305	467	321
Karlstad				0	247	243
Göteborg					0	149
Jönköping						0

On debugging
Those of you that really want to do some programming yourselves will be greatly helped if you can debug the code. Debug means to remove errors but you can also use it to step through code one line at a time and follow how variable values change. In RStudio you can debug your code as in any other programming software. To do this you have to 'debugSource' the code.

1. Download the code GA.R from the course website and store it in a suitable directory.

2. Put a breakpoint somewhere in the beginning of the code, for example at line 5. You do this by clicking the grey margin to the left of the statement `pop_size <- 50`. If a red dot appears you have succeeded to make a breakpoint.

3. Click the 'Source' button, choose 'Source' from the menus, or use 'debugSource('GA.R') in the console window.

4. A breakpoint means "stop here" when code is executed. This means that the execution of GA should have stopped at line 5. This is indicated by a green arrow in the left margin. The whole line that is to be executed next should be highlighted in blue.

5. If you click "Debug" in the menu you can see the debugging options. For example F10 will execute one line at a time. If you press F10 you can see under "Values" how pop_size is created and set to the value 50. You should also see buttons in the console window for some of the functions.

6. When you get down to line 17 the whole function gets highlighted in yellow. Now, this may be a bit confusing, but you cannot check how Create_pop actually works here because the function is not really used until line 107. Right now it is only being defined, which creates the Create_pop function object.

7. Step through the code for all functions until you reach where the execution of the main program starts.

8. Now you can press Shift + F4 to enter the Create_pop function and then step through it line by line with R10.

9. If stepping through create_pop takes too long time an gets boring you can leave it with shift + F6.

10. Try stepping into and out of the other functions!