# Artificial Intelligence 2: Comments on worksheets 1 & 2

**Recap on search algorithms:**

Last year in Artificial Intelligence 1 we talked about problem solving as search through a space of candidate solutions for the ‘best’ one. We also looked at a several ‘single member’ search algorithms such as depth/breadth first, A\*, best-first, and a simple hill-climber (also called local search). All these had in common that they considered one solution at a time (the workingCandidate) and moved from solution to solution- getting better all the time.

**Basic idea of an evolutionary algorithm (EA):**

Evolutionary Algorithms also work in a series of iterations, but the principle difference is that they use a number of candidate solutions – which because of the biological inspiration they call a ‘population’ of individuals.

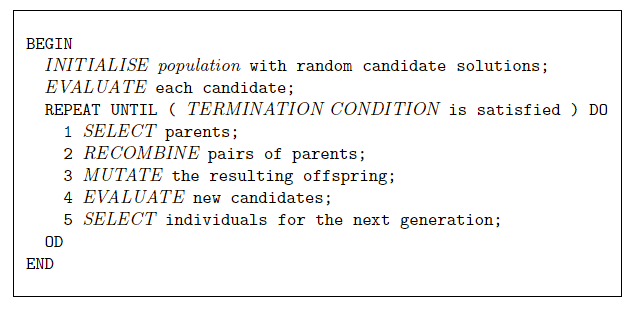
Thus the python class Larry gave you called individual is the same as the ‘candidateSolution’ class we had in AI1. Also, the array of ‘variableValues’ tends to be called ‘genes’ – but they both serve the same purpose: each unique combination of gene values specifies a different candidate solution to the problem we are trying to solve.

In these worksheets and the coursework you will implement a simple type of Evolutionary Algorithm called a Genetic Algorithm - or GA for short.

The simplest and most straightforward way to implement this just uses:

* a few global variable constants for the population size, number of genes (or decision variables), and the number of generations to run for
* Two arrays of individuals called “population”, and “offspring” –with global scope
* We will import the python library function random and use two functions from it:
  + Random.random() and random.randInt() – look these up online
* Sometimes (for example, during selection) we will want to make a copy of an object A that we can change without changing the original. There are two ways to do this:
  + Import the library ‘copy’ and use Acopy = copy.deepcopy(A)
  + Create a new object Acopy of the same type as A, and then explicitly copy across the contents of A into Acopy.
  + Larry’s code gives you examples of both of these ways of doing it.

The pseudocode for the GA looks like this:



So your python is going to end up looking something like this:

import copy, random

N = 10 #number of genes – keep small during debugging to fit on screen

P = 10 # size of population – again keep low during debugging

NUMGENS = 10 #number of generations

class individual:

def \_\_init\_\_(self):

self.gene = []

self.fitness = 0

population = [] #declare with global scope -as empty array to start with

offspring = [] #declare with global scope -as empty array to start with

# code to calculate then return the fitness of an individual

## of course this will be different for every different problem

def calculateFitness(ind):

fitness = 0

# code here

return fitness

# INITIALISE

##code to fill population with P objects of type indiv. Each has its N genes set to 0 or 1 at random

#EVALUATE

# loop calling fitness function to set fitness for each population member

#this is the main evolution loop

for gen in range(NUMGENS):

offspring = [] #discard the previous offspring

#SELECT

## loop P times selecting a member of the population,

##and putting a copy in offspring

#RECOMBINE

## code that performs the crossover on each pair of offspring

#MUTATE

## loop through each gene of each offspring,

##changing gene value with a given probability

#EVALUATE

## same code as to start, but for offspring not population

#SELECT indivs for next generation: a simple GA uses all the offspring

## code to replace contents of the population with the offspring

#end of evolution

## code to display results

HINT:

It is a good idea to have a few lines of code that quickly works out and prints the average and best fitness in either the population or the offspring. If you store these values in an array, then you can make progress graphs after the GA has finished. Remember there is nothing to stop you adding a print() method to the class *individual*

**Activity 1: INITIALISATION**

Based on the template above, enter the code to initialise the population, then evaluate it. **To test your code**:

(i) add a loop to print each member, make sure its genes look random and that the stored fitness is correctly calculated as the number of genes with value 1.

(ii) check that the average fitness is N/2 - and that you understand why this is!

**Activity 2: SELECTION**

Now add code to implement the selection operator that makes P choices from the parents and makes copies of those parents in the offspring array.

Also put in code to implement the EVALUATE call within the loop so that you know and can display the offspring fitnesses.

**To test your code**:

Start with NUMGENS=1. The offspring average fitness should be higher than the population average, the best fitness will usually be the same: never higher in the offspring, but occasionally lower.

If in doubt, print the offspring and make sure each is a copy of something in the population.

Then set NUMGENS= 50: you should see that over time the selection operator makes more and a more copies of the best in the population until it takes over, so the average fitness = best fitness.

**Activity 3: RECOMBINATION**

Now add code to implement the recombination operator which randomly swaps sets of genes between pairs of offspring. This is explained in the lecture slides. You are given some example code that implements ‘one point crossover’, and there are lots of descriptions on the web to help you.

**To test your code:** you could run for one generation and print out each pair of offspring before/after crossover. If you run for more generations, you should see the best rise for a while, then ‘stall’, and the average fitness rapidly rise to meet it as the population again gets taken over by copies of the best individual discovered.

**Activity 4: MUTATION**

Add code to implement the mutation operator. HINT if the genes just hold 1s and 0s, you can do this ‘in place’ by saying *if (random.random()<MUTRATE): offspring[i].gene[j] = 1- offspring[i].gene[j]*

**To test your code:** Start with MUTRATE = 1.0/N. When you have selection-recombination and mutation working together your GA should rapidly evolve a solution with the optimal fitness (N) - but the population average fitness will remain lower as most mutations introduce errors rather than ‘happy discoveries’. If you increase the probability of mutation to 50%, the extra randomness should make evolution ‘stall’, and if you set the probability to 0 it should behave the same as in activity 3.

**Activity 5: INVESTIGATION**

Import the matplotlib.pyplot library and use it to plot the best and average fitness on the y axis with generation on the x axis. (HINT we used this library a lot in weeks 5-9 of AI1, and you just need the plot() function).

Experiment with commenting out different combinations of the selection, recombination and mutation code and see if you can explain what the effect is on evolution, and why