# Evolutionary Computation – Practical Assignment 1

**Author:** Tom Ormsby (t.ormsby@students.uu.nl)  
**Date:** 11/03/2018

## Abstract

In this assignment, we seek to compare the performance of a simple Genetic Algorithm (without mutation) on a combination of fitness functions including: Counting Ones, Deceptive & Non-Deceptive Trap, and Deceptive & Non-Deceptive Trap with Random Linking. Each function was tested with both Uniform and 2-Point Crossover methods, and key statistics computed. The results show that the simplified Genetic algorithm performs best on the Counting Ones function and shows that 2-point crossover can improve the speed of convergence on a solution.

Significant performance decreases were noticed as the complexity of the fitness function increased. The introduction of random-linking led to unmanageable computing times when running on a single core, and I was forced to adopt a multi-core approach to allow for the experiments to run.

## Introduction

I chose to build my solution in Python 2.7, using an Anaconda distribution and the Spyder IDE. I set up a virtual environment, with requirements output to **requirements.txt.** Results were built into nested lists, that were then ‘pickled’ in the working directory for recall. The end program was run on my desktop, with **<CPU SPECS>:**

The scope of this projects required us to run two sets of experiments, one searching for the minimum population size to achieve an optimal solution in 24 out of 25 runs, and one to measure various features of the population over consecutive generations for a population size of 250.

## Solution Design

The solution design was clouded by my need to learn Python as I was producing it, and as such some of the design has proven inelegant/inefficient at best and fails to utilise some of the strengths of Python. The final solution did not consider what data was needed for each experiment, and so passes a large volume of unused data between functions.

I split the Genetic Algorithm into key sections and designed functions to deal with them.

1. Generating the Population

A population of randomly generated binary strings was needed for each run of the algorithm. I used the ‘*random’* library in *numpy* to generate an array of binary strings with each row representing a member of the population.



The RNGs within python derive their default state (without a seed) by observing the clock at the point the process is initialised. In the case of multi-core processing, this led to a novel situation where multiple processes are initialised at the same time, and thus inherit the same random state. Within the context of a multiple trial experiment requiring 25 random populations, this was not ideal! I introduced a seed variable that was passed and ensured each trial had a different random population.

In later generations the population is shuffled before any of the crossover and selection rules are applied.

1. Calculating Fitness for a Member

The fitness of a population member was needed to determine selection. Although we had 5 separate fitness functions, the 4 Trap Functions were variations of themselves. I decided to build the fitness calculation into one function that took arguments to determine which final function to use.



The function takes in a number of variables and was coded to allow the tailing of a choice of **d** and **k** for further exploration, although this functionality was not explored in this assignment.

A further helper function was built to apply this across a population.

1. Performing Crossover on two Members

Crossover operations are the driver of the creation of new solutions. In this assignment we consider the simple Uniform Crossover, and an expansion on this with 2-Point Crossover. The crossover function takes in two members of the population, alongside a number of ‘**cross\_points**’. The function returns two children created by performing crossover at a randomly generated point (or multiple randomly generated points) along the string.



We create the range of potential points to crossover by evaluating the length of the string. We are taking the crossover point **n** will cut the string immediately before that index. As such we generate the range of available cross points as *range(1,len(p1))*.

As the crossover process is iterative for multi-point crossover, this is coded within a loop. An alternating Boolean is used to track which parent the next subsequence is being pulled from.

A helper function was created to apply this across a population by pairing consecutive members.

1. Selecting the Members for the Next Generation

A family competition is used to determine which 2 of the parent and children pass on to the next generation. When fitness is equal, children are preferred for selection over their parents. A simple list sort allowed these rules to be satisfied.

A further condition of the algorithm was to end processing when no children reach the next generation. A global variable *continue\_algorithm* is accessed within the function to determine this. It is set to False immediately before applying the selection across the parent and child populations, and is turned True if any child is selected.



A further requirement of this function was to track if a selection error occurs. This is defined as when both children have a 0 in a particular position of the string, but one of the parents contains a 1 in this position.

1. Further Considerations

The algorithm was said to finish when no children made it to the child population. As our selection function prioritises children when parents are equal in fitness, the algorithm would not stop if it converged. Additional fail-safes were placed to terminate the algorithm if 5 consecutive generations were identical, or if the generation count reached 1000.

## Results

## Conclusions