# 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.

## 1. 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 an *AMD64 Family 21 Model 48 Stepping 1 AuthenticAMD ~1900 Mhz* processor, with 4 cores.

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.

## 2. 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.

### i. 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.

```
#General Usage
pop = np.random.randint(2,size=(pop_size,string_length)


#Usage within multi-core process
local_state = np.random.mtrand.RandomState(seed)
pop = local_state.randint(2,size=(pop_size,string_length))
```

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.

### ii. 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.

```
#x is a population member, further args indicate the function to use.
#Default values are set up but can be tailored on call.
def fitness( x, trap = False , random_linked = False, k = 4, d = 1.0 ):
    fitness_value = 0

    #If trap is called, push into the trap function.
    #Variable y is assigned as random linking uses shuffle(), which acts on
    # an item in place and would change the member.
    if trap:
        y=x.copy()

        if random_linked:
             random.shuffle(y)

        #Trap Function Evaluation
        for j in range (0,int(len(y)/k)):
            if sum(y[j*k:(j*k)+k])==k:
                fitness_value += k
            else:
                fitness_value += ( k - d - ((k-d)/(k-1)) *
                                                    sum(y[j*k:j*k+k])
        return fitness_value
    #Evaluate Counting Ones Here (if not trap)
    else:
        return sum(x)
```

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.

### iii. 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.

```
#Function takes two parents as arguments and a number of cross_points.
def crossover( p1, p2, cross_points = 1):
    #Cannot cross before first or after last point in string
    #All cross points must be at least one bit apart
    #Crossover points occur before the given index
    #Crossover points generated from sampling the no. of available points
    cross_locations = sorted(random.sample(range(1,len(p1)), cross_points))

    #Add the end of string as final cross point or the sub-sequence does
    # not run up to the end of the string
    cross_locations += [len(p1)]

    #boolean to store which parent we're buildign from for multi-crossover
    first_parent = True
    #Initialise start point of subsequence
    last_crossover=0
    c1=[]
    c2=[]

    #Cycle through the generated crossover points to complete children
    for cross_point in cross_locations:
        if first_parent:
            c1 += p1[last_crossover:cross_point]
            c2 += p2[last_crossover:cross_point]
        else:
            c1 += p2[last_crossover:cross_point]
            c2 += p1[last_crossover:cross_point]
        first_parent = not first_parent
        last_crossover = cross_point

    return [c1,c2]
```

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.

### iv. 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.

```
#Takes in two lists of tuples of member and fitness
def family_competition(parents,children,continue_algorithm):
    #combine family, children first so sort prioritises these.
    #sort by descending fitness and take the first 2.
    selection_error = 0
    family = children + parents
    family.sort(key=itemgetter(1),reverse=True)

    #Selecting the top 2
    for member in children:
        if member in family[0:2]:
            continue_algorithm=True

    #Running a check for selection errors. Look for values of selected that
    # are both 0 (at that index), and store the indexes
    selection_check1 = set([i for i, x in enumerate(family[0]) if x == 0])
    selection_check2 = [i for i, x in enumerate(family[1]) if x == 0 and i
                                                        in selection_check1]
    #Compare values of parents at these indexes. If either is 1 then
    # increase selection error counter.
    for i in selection_check2:
        if parents[0][i] == 1 or parents[1][i] == 1:
            selection_error=selection_error+1

    return (family[0:2],selection_error)
```

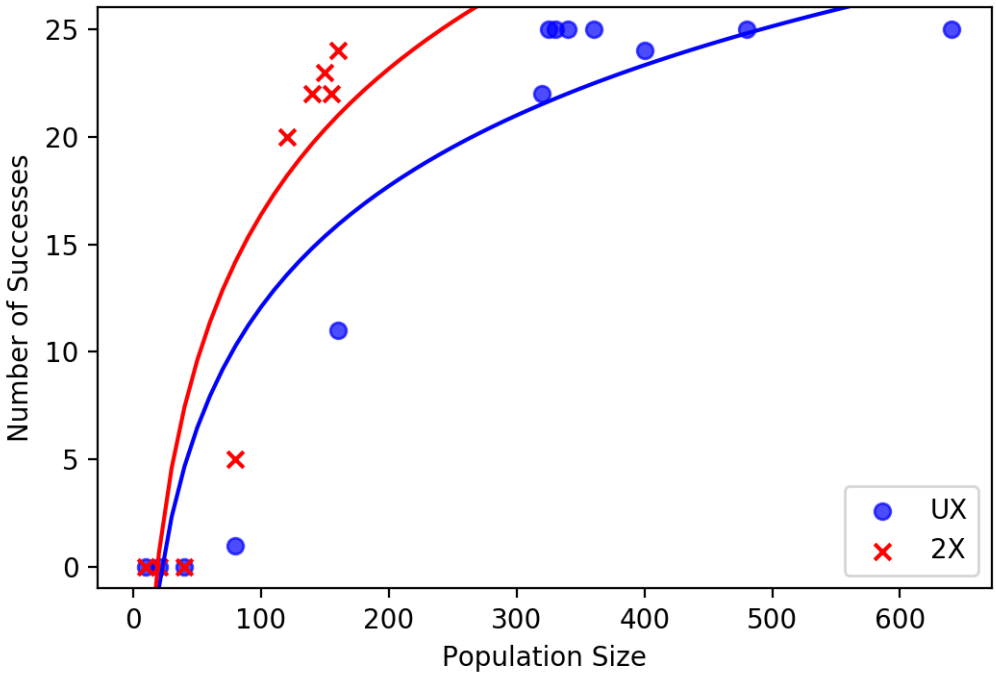
A further requirement of this function was to track if a selection error occurs. This is defined as when both children have a 0 in a particular position of the string, but one of the parents contains a 1 in this position.

### v. 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.

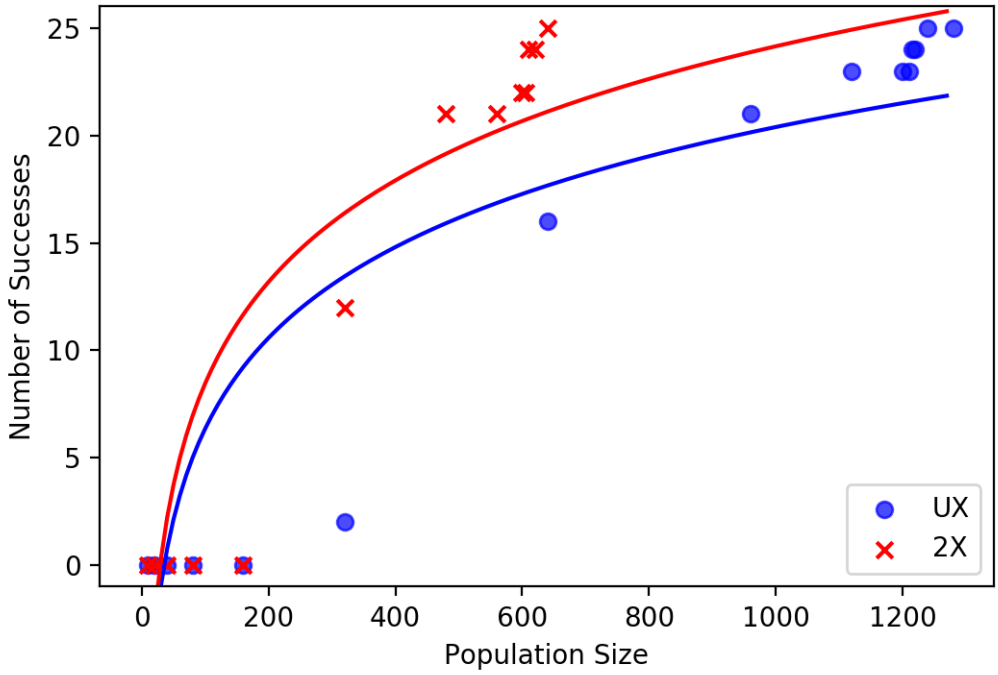
## 3. Results

### i. Minimum Population Experiments



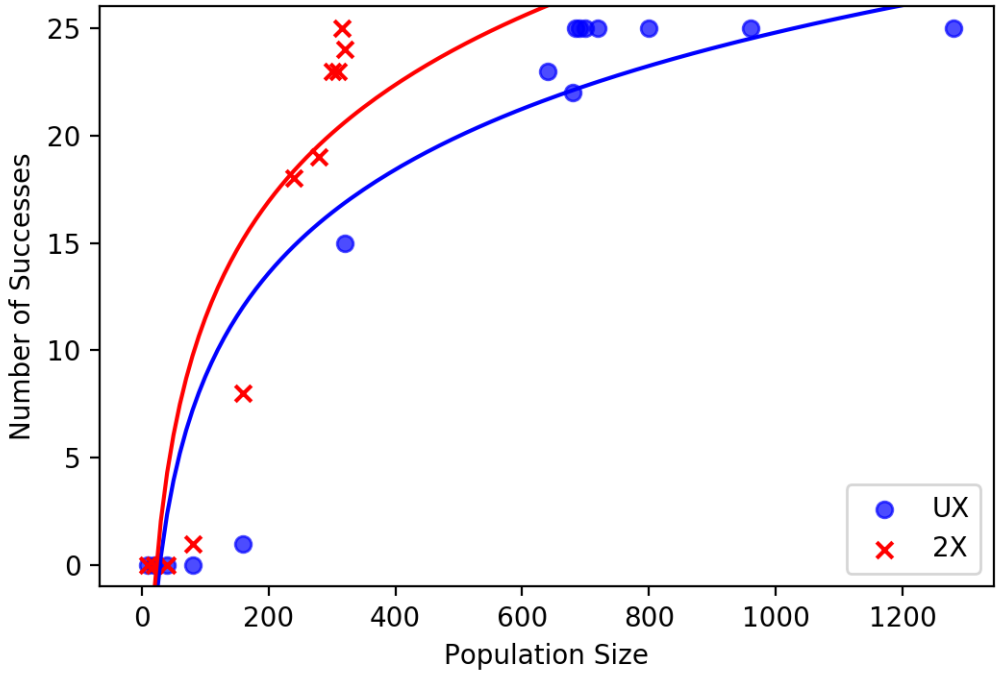

| | Min_Pop | Avg_Generations | Avg_Fitness_Evals | Avg_CPU_Time |
|---|---|---|---|---|
| 2X | 160.0 | 50.36 | 8057.6 | 0.7118763065338135 |
| UX | 330.0 | 53.0 | 17490.0 | 1.4412523078918458 |

**Fig 1 – Counting Ones Performance with UX & 2X**



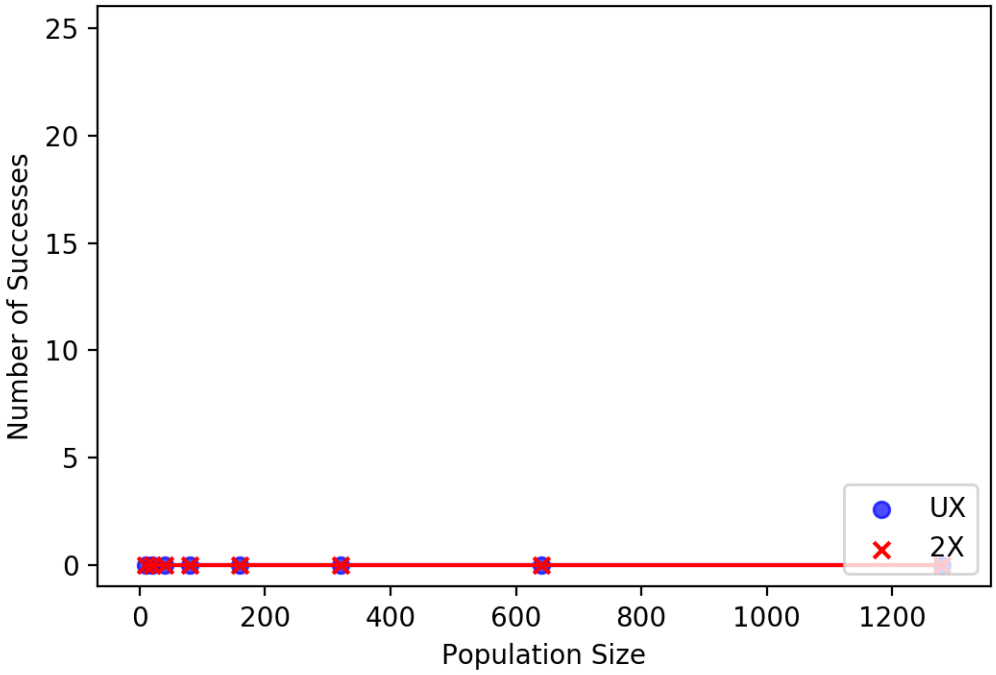

| | Min_Pop | Avg_Generations | Avg_Fitness_Evals | Avg_CPU_Time |
|---|---|---|---|---|
| 2X | 610.0 | 56.6 | 34526.0 | 9.540634431838988 |
| UX | 1220.0 | 57.36 | 69979.2 | 19.79189525604248 |

**Fig 2- Deceptive Trap Function Performance with UX & 2X**



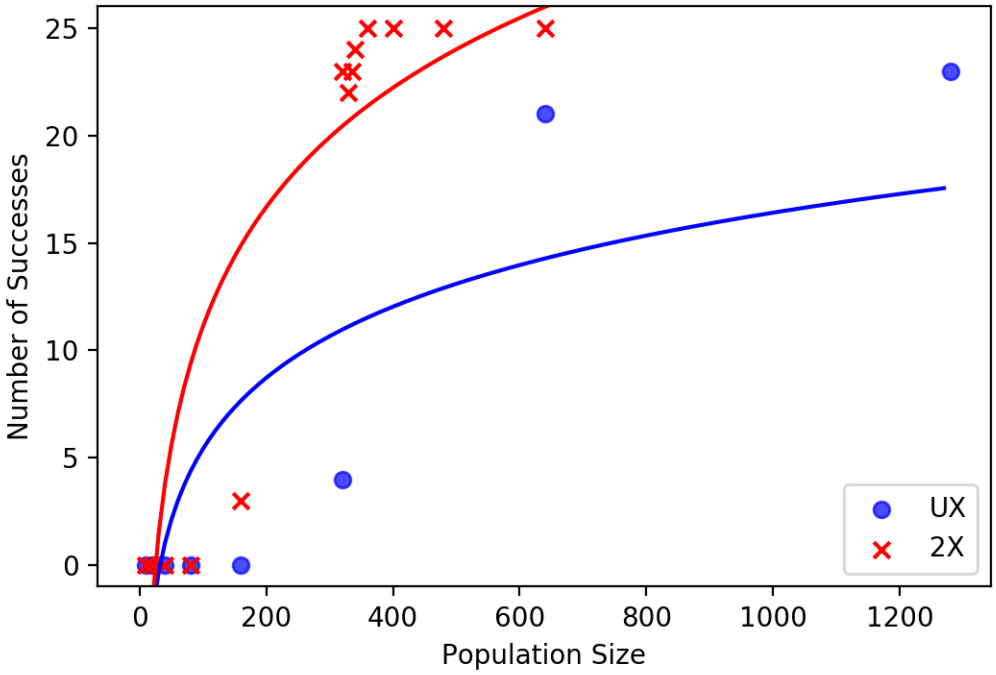

| | Min_Pop | Avg_Generations | Avg_Fitness_Evals | Avg_CPU_Time |
|---|---|---|---|---|
| 2X | 320.0 | 49.12 | 15718.4 | 4.318130331039429 |
| UX | 610.0 | 50.2 | 30622.0 | 9.297509279251098 |

**Fig 3 – Non-Deceptive Trap Function Performance with UX & 2X**




| | Min_Pop | Avg_Generations | Avg_Fitness_Evals | Avg_CPU_Time |
|---|---|---|---|---|
| 2X | None | None | None | None |
| UX | None | None | None | None |

**Fig 4 – Randomly Linked Deceptive Trap Function Performance with UX & 2X**




| | Min_Pop | Avg_Generations | Avg_Fitness_Evals | Avg_CPU_Time |
|---|---|---|---|---|
| 2X | 340.0 | 138.04 | 46933.6 | 22.375647172927856 |
| UX | nan | nan | nan | nan |

**Fig 5 – Randomly Linked Non-Deceptive Trap Function with UX & 2X**

ii. Counting Ones Measures

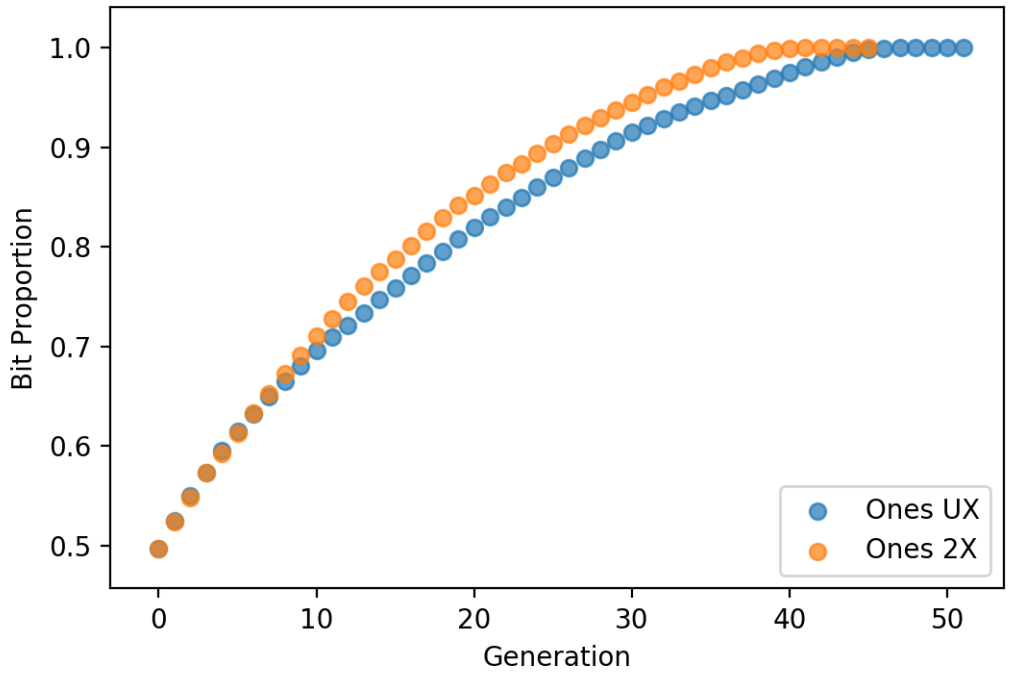



**Fig 6 – Bit Proportions of Counting Ones for UX & 2X**

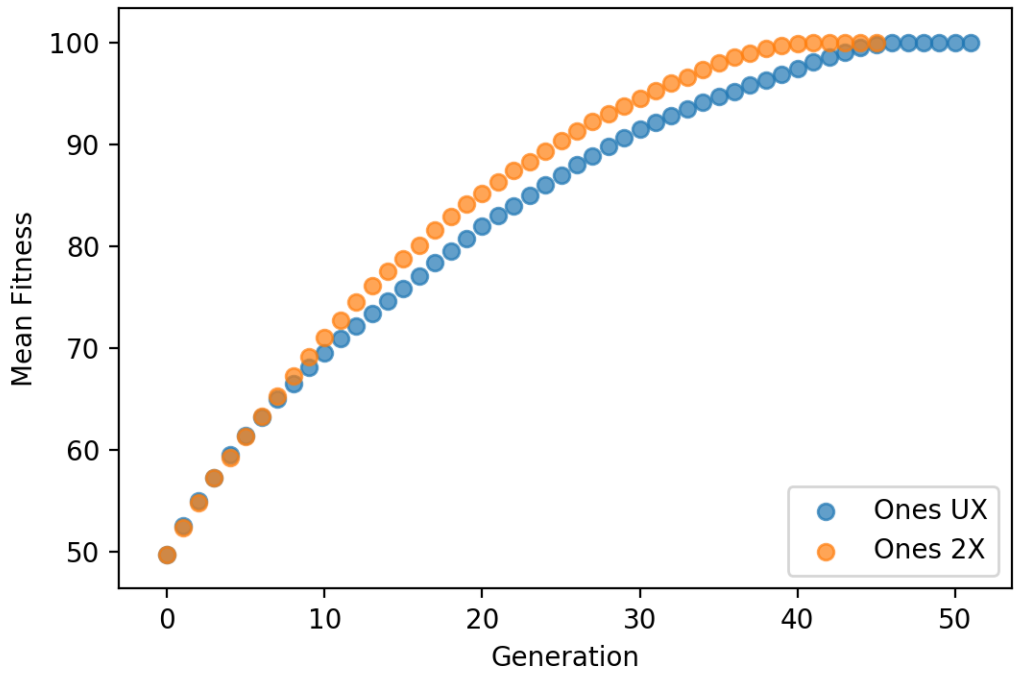



**Fig 7 – Mean Fitness of Counting Ones for UX & 2X**

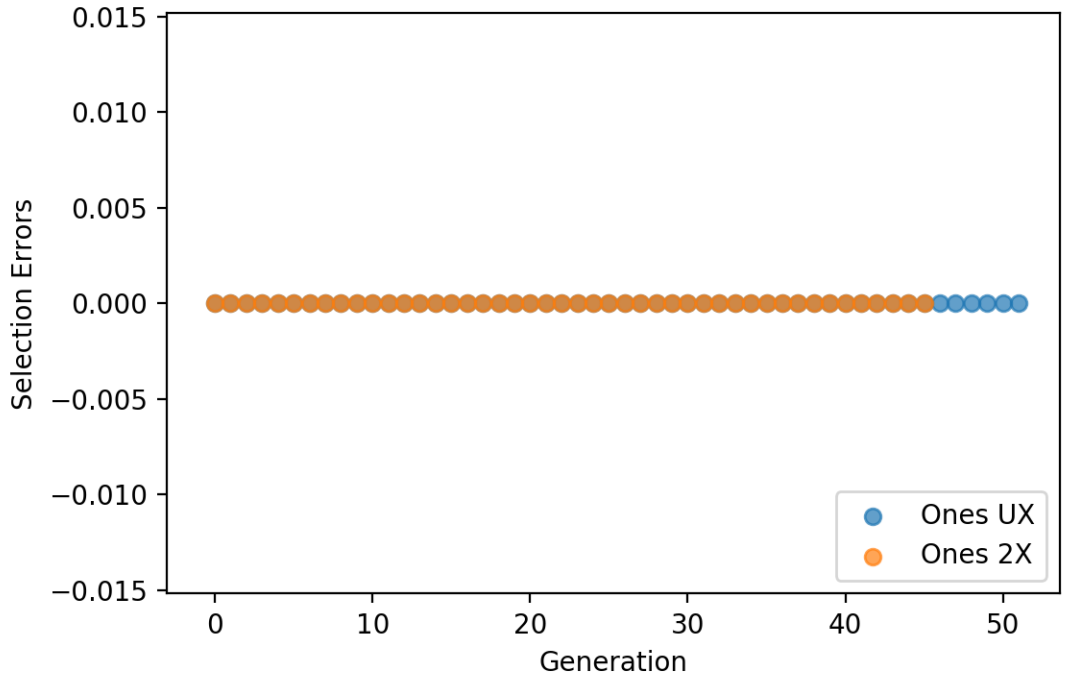



**Fig 8 – Selection Errors of Counting Ones for UX & 2X**

**NB:** There appears to have been a bug in passing my 'Selection Errors' data, resulting in values of 0 for every generation. I would expect instead a decreasing number of selection errors, as the generations progressed. This fall-off should be slightly sharper for the 2X graph as it converges quicker.

## 4. Conclusions

Within the course of this assignment, I have learnt a great deal about programming in Python and managed to obtain some interesting results from applying a simple Genetic Algorithm to 5 fitness functions. Some observations and learnings are considered below.

### i. Computing Time/Cost

As the fitness functions used become more complex, the time spent to compute solutions significantly increases, which is especially noticeable when working with the Randomly Linked functions. These required nearly ten times as long to evaluate as the Counting Ones function.

My initial attempts to run the experiments on a single core led to Experiment run times of several hours, and I instead had to pick up multi-processing knowledge to make the task manageable.

It can also be observed that the 2X algorithms run in roughly half the CPU time of their UX alternatives, though this is largely down to the smaller minimum population sizes.

### ii. Generations & Fitness

The number of generations required to reach an optimum for the Trap and Counting Ones functions averaged out at around 50 for both UX and 2X and appears to be influenced more by the string length than the fitness function.

The introduction of Random Linking however made it much harder to find solutions, as it added noise to the information being gathered on each member, and lead to sub-optimal selections. For the Deceptive Trap function, random linking resulted in neither the UX or 2X methods having any successful trials up to a population size of 1280, and the UX method was unable to reach 24 out of 25 successes with the Randomly Linked Non-Deceptive Trap also.

The difference in performance between the Randomly Linked Deceptive Trap and Randomly Linked Non-Deceptive Trap is likely a function of the fitness values. With random linking on the Deceptive Trap, a string with a large proportion of 0s is likely to form many ‘0000’ blocks, and receive a high fitness for them, meaning it can challenge for selection against a block with more 1s, but without many ‘1111’ blocks formed from random linking. Once this is scaled down in the Non-Deceptive Trap function, more information appears to be retained by the next generation, allowing a convergence towards the optimum.

### iii. Population Sizes & Multiple Optima

In the simplest interpretations of our fitness functions, we have a solution space with one optimum (Counting Ones), or two optima (Trap Function). When dealing with a single optimum, the function converges with a smaller a population, however the introduction of a Local Optimum as in the trap functions requires a higher population to guarantee convergence on the Global Optimum.

If the fitness of this Local Optimum approaches the Global Optimum, we require a much higher population to guarantee the Global Optimum is chosen. This can be seen in the differences between the minimum populations for the Deceptive and Non-Deceptive Trap. The fitness of the Deceptive Trap’s local optimum is twice as high as that of the Non-Deceptive Trap, and as such the minimum population required was also around twice as high.

### iv. Solution Design and Other Take-Aways

The assignment challenged me to expand my knowledge, and I was (somewhat) successfully able to pick up a programming language, and design a functional solution however the performance and usability of this was hindered by my initial knowledge. If I were to return to this problem in the future, I would consider a multi-core approach from the beginning. The Genetic Algorithm approach lends itself well to parallelisation, and this could really improve performance over the repetitive calculations.

A real weakness of my solution is the construction of data structures. As I continued to build helper functions to apply the core functions across a population, my data became nested deeper and deeper in a maze of Lists and Tuples. This in turn made it hard to recall, and near impossible for anyone picking up the code to easily rationalise. A better approach may be to consider separating some of the functions out, using python dictionaries, or taking a more Object-Oriented approach.