# IT3708 Project 2

Programming an Evolutionary Algorithm (EA)

This report is written for the second project in the course IT3708 Sub-symbolic AI Methods at the Norwegian University of Science and Technology during the spring semester of 2016. The task was to implement a basic evolutionary algorithm library and use it to first solve two smaller problems (OneMax and LolzPrefix) and then later find surprising sequences

I did not manage to complete the surprising sequences part of the project, so surprising sequences is not mentioned in this report.

## Architecture & implementation

This project has been implemented in Python 3.5 and is dependent upon two libraries in order to run; numpy, which is a scientific computing library, and matplotlib for plotting the data.

The main class of my implementation is the *EA* class, which keeps track of a *Population* and iterates through the evolutionary loop by calling *population.breed()* for every generation. This class is also responsible for logging and plotting data points from every generation. A *Population* object is initialized with a *mate selection method*, an *adult selection method* and a *phenotype*. It manages a pool of children and a pool of adults, and how the population should evolve based on constructor parameters. Currently, my implementation only supports one type of genotype – that is a bit vector. Genetic operators such as crossover and mutation are class methods of a Genotype object. Phenotype objects take a *genotype* as a parameter, and the three sub-classes *OneMaxPhenotype, OneMaxRandomPhenotype* and *LOLZPrefixPhenotype* are initialized by calling a factory method on the super class. New subclasses are easily created by overriding the *fitness* and *develop* methods. *Adult selection* is done in a similar fashion; there exists a superclass *AdultSelection* of which *FullGenerationalReplacement, OverProduction* and *GenerationalMixing* inherits from. The subclasses must override the *select* method. Subclasses are easily instantiated by calling a static factory method on the super class. Mate selection methods such as *FitnessProportionate, SigmaScaling, Boltzmann* and *Tournament* inherits from *MateSelection* and must override *select* and *expected_value*. Again, subclasses are instantiated by calling a static factory method on the super class.

Running the project is done through a simple input loop, where the inputs are saved to a globally accessible configuration module which the different classes use.

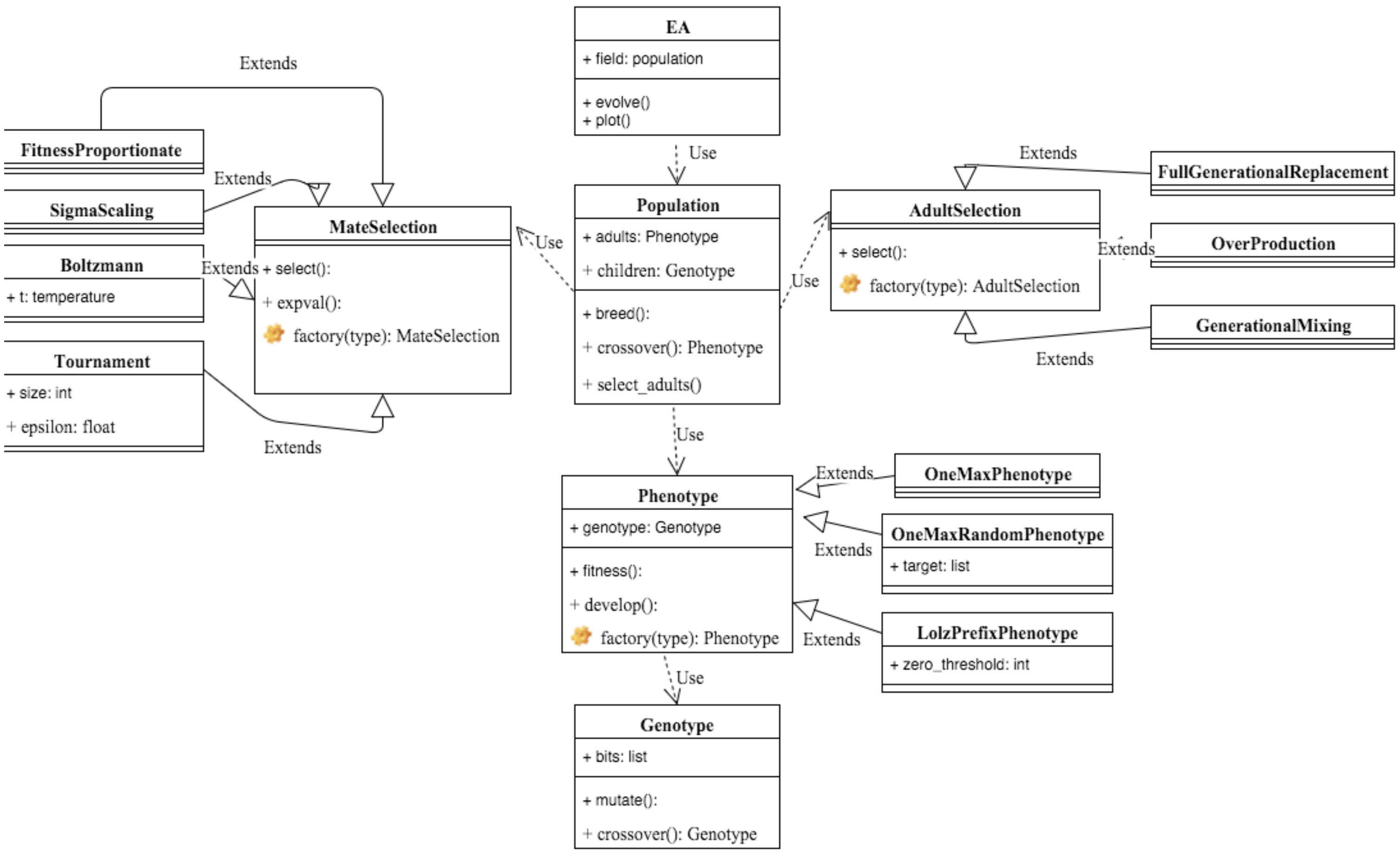


*Figure 1: Overview of the architecture*

By using this architecture, handling new types of problem is easy; you can simply subclass Phenotype, MateSelection or AdultSelection, based on what is needed. Below is an example of how you would subclass Phenotype:

```
class OneMaxPhenotype(Phenotype):
    def __init__(self, genotype):
        Phenotype.__init__(self, genotype)

    @property
    def fitness(self):
        return sum(self.genotype.bits) / len(self.genotype.bits)

    def develop(self, genotype):
        return genotype
```

## Analysis of performance on the One-Max problem

I initially started with mate selection method *FitnessProportionate* and adult selection method *FullGenerationalReplacement*, mutation rate of 0.01, crossover rate of 0.8 and varying the population size from 200 up to 500 (*see figure 2.1 and 2.2)*.

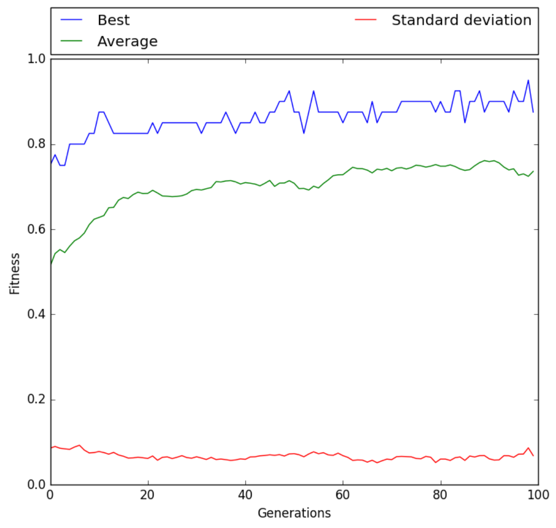


*Figure 2.1: mutation 0.01, xo 0.8, pop 200*

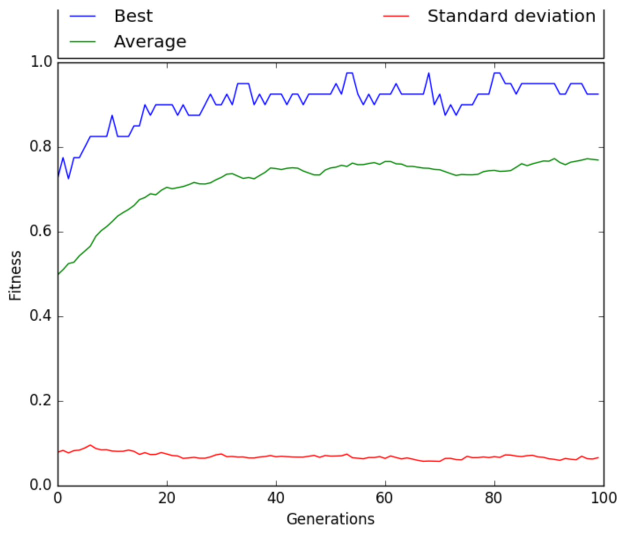


*Figure 2.2: mut 0.01, xo 0.8, pop 500*

After multiple runs with population size 500, I decided to change the mutation rate to 0.001 because I couldn't reach the desired target fitness within 100 generations. With the new configuration, the target fitness was consistently reached. I then experimented with different crossover rates while continuously lowering population size.

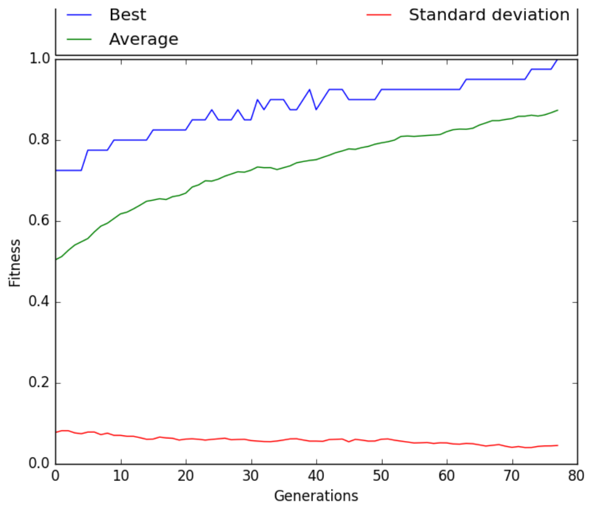


*Figure 3.1: mut 0.001, xo 0.8, pop 500*

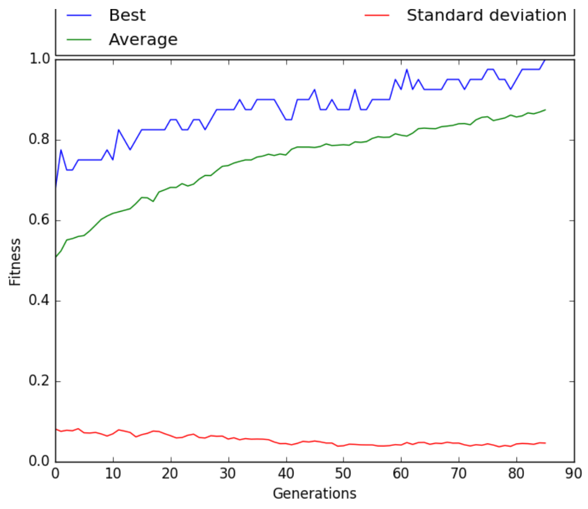
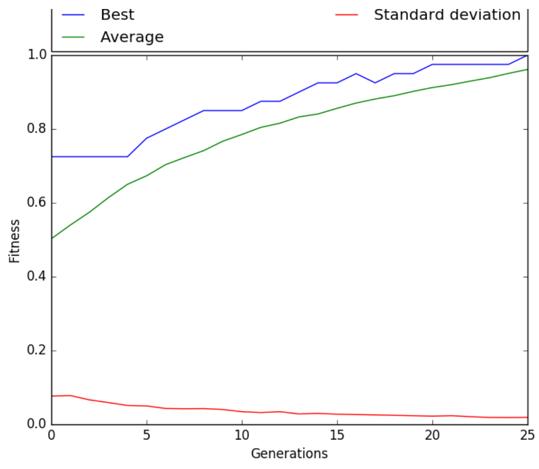


*Figure 3.2: mut 0.001, xo 0.9, pop 250*

**Different mate selection**

Still using population size of 250, mutation rate of 0.001 and crossover rate of 0.9, I tried changing the mate selection method to *SigmaScaling*. This yielded much better results, with the EA usually reaching target fitness in less than 30 generations (figure 4.1). *TournamentSelection* with *e = 0.2* and *size = 6* yielded even better results (figure 4.2).



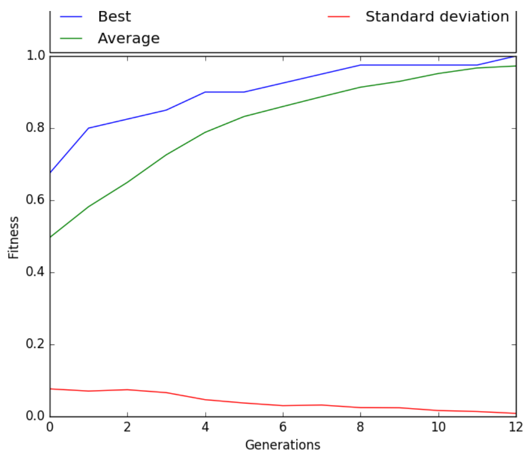
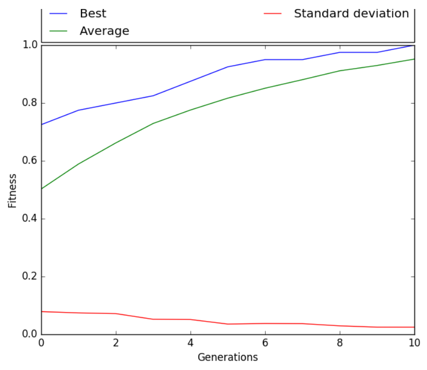
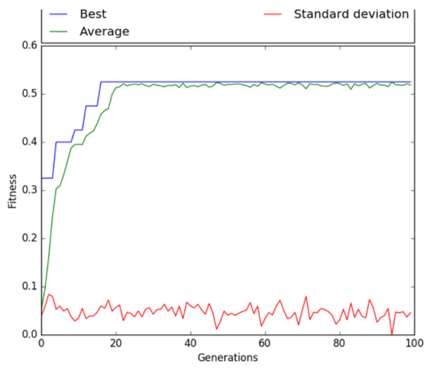


*Figure 4.1*

*Figure 4.2*

**Difficulty of OneMaxRandom**
There is no change in difficulty by changing the target vector of only ones to random bits – the evolutionary algorithm does not care what the target vector looks like. By doing 4 runs, I achieved approximately the same results as for regular OneMax (see figure 5)





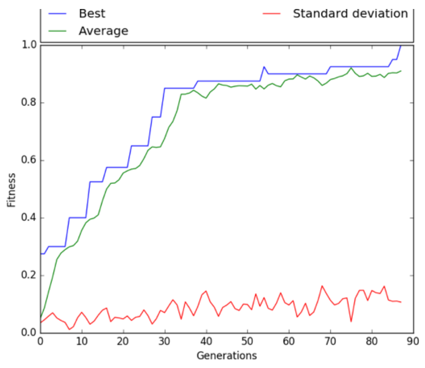


*Figure 5*

*Figure 6.1*

*Figure 6.2*

**The LOLZ Prefix problem**
In this problem, the AI will generally reach one of two states; either it gets stuck at a local optimum with a very slim chance of ever improving (figure 6.1), or it will reach the desired fitness within a reasonable amount of generations (figure 6.2). If the EA is focusing on leading ones, it will reach the target fitness. If the EA focuses on leading zeroes, it will converge on a fitness of 0.525.

**Difficulty of the problems**
From the perspective of an evolutionary algorithm, the OneMax problem (both regular and randomized) is the simplest because there exists no local optimum. The LOLZ prefix problem is harder because of the existence of local optimas when there are leading zeroes.