IT3708: Project 2 – Programming an Evolutionary Algorithm (EA)

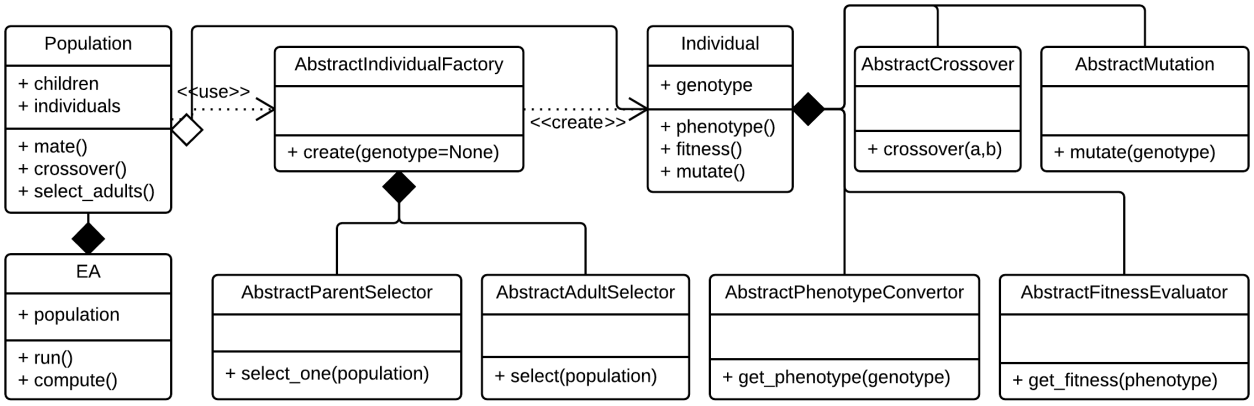
Author: Petr Zvonicek

a) Implementation

EA is implemented in Python using modules *numpy* for efficient computation, *matplotlib* for plotting and *PyYAML* for configuration management.

The entry point is the class *EA*. This class is responsible for initializing the population and the basic evolutionary loop. For each generation it calls adult selection, mating and mutation on *Population* object. *EA* also provides plotting and logging logic. *Population* class manages adults and children and handles their evolution. Adults and children are represented by *Individual* class. This class has *genotype* property and provides a way to mutate it, convert it to phenotype and compute fitness.

The concrete algorithms for crossover, mutation, phenotype conversion, fitness computation, parent and adult selection are specified using the GoF Strategy design pattern, which ensures modularity of the system. Individuals are created using the GoF Abstract Factory design pattern. This makes the system more extendible and allows, for example, to modify the way the new genotypes are generated.



Modularity

The EA is configured via the globally accessible module *config*. This module contains configuration data extracted from *YAML* configuration file which must be provided as an argument on launch. This configuration file is holding all the configurable parameters such as population size, type of EA, adult/parent selector, phenotype convertor, fitness evaluator, mutation and crossover strategies etc. Example of a part of configuration file is below.

```
population_size: 20
generation_limit: 200
...
adult_selector: !!python/object:ea.adult_selection.FullGenerationalReplacementAdultSelector {}
parent_selector: !!python/object:ea.parent_selection.SigmaScalingParentSelector {}
fitness_evaluator: !!python/object:ea_impl.one_max_problem.OneMaxFitnessEvaluator {}
```

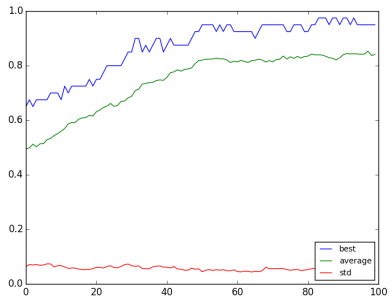
Implementing new AI is therefore just a matter of subclassing *FitnessEvaluator*, *Individual Factory*, *PhenotypeConvertor* (and possibly other classes if needed) so as to provide a new strategy and creating a new *YAML* configuration file with desired parameters. Example of a fitness evaluation strategy is below.

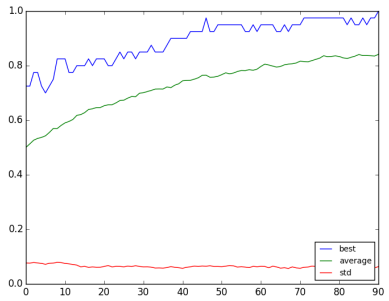
```
class OneMaxFitnessEvaluator(AbstractFitnessEvaluator):
    def get_fitness(self, phenotype):
        ones = 0
        for bit in phenotype:
            if bit == 1:
                ones += 1
        return ones/len(phenotype)
```

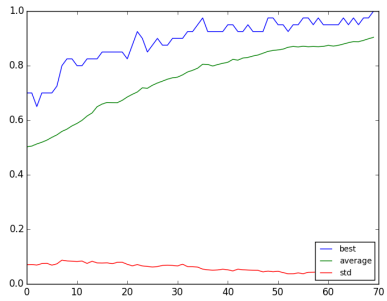
b) One-Max problem

40-bit vector, adult selection: full-generational replacement, parent selection: fitness-proportionate

At first, I run the simulation with **mutation rate 0.01** and **crossover rate 0.7**. The both average and maximum were converging to the fitness 1.0, smaller populations (<100) did not performed very well though and usually did not exceeded fitness 0.95 (first figure). I needed to go up to the **population of 300**, which gave me reasonable probability the fitness 1.0 will be reached within 100 generations. (second figure). Upon this population, I tried to experiment with mutation and crossover rates. Lowering the **mutation rate to 0.001** seemed to find the desired fitness in fewer generations. Tackling with crossover rate seemed to provide only small differences, but increasing it to 0.9 performed slightly better.
After these tweaks, I was able to **lower the population from 300 to 170** while preserving the reasonable probability of reaching the fitness 1.0 within 100 generations. (third figure).

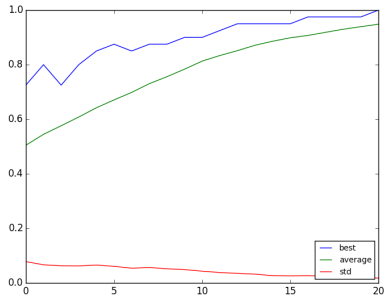


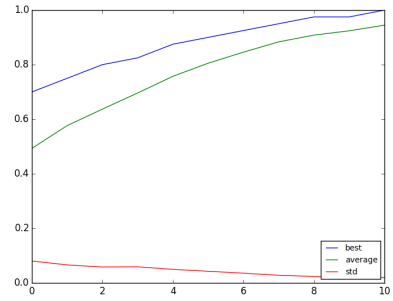




Experimenting with parent selection

Using the previous configuration, I've tried to use **sigma-scaling** instead of fitness-proportionate. The difference was enormous. Desired fitness was now reached within just 20 generations instead of 80-100 when used fitness-proportionate (first figure). Even better results were provided by **tournament** parent selection with group_size = 5 and epsilon = 0.1. The target was now reached usually within just 10 generations (second figure).





Modified target bit string

Modifying the target bit from all 1's to a random bit doesn't change difficulty of the problem. For AI it is just one arbitrary bit vector which needs to be found. Following chart document this. I conducted 8 runs with random

target vector (blue bars) and all 1's vector (green bars). There is no visible difference between these two targets, which confirms my assumption.

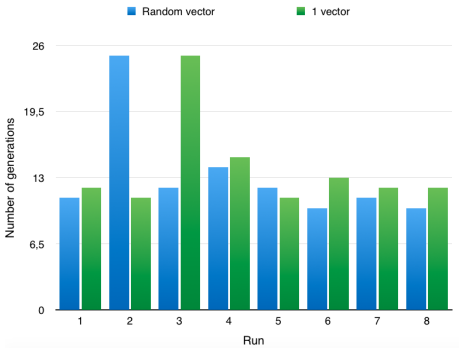


Figure 1:

LOLZ

Results can be divided into two cases – either the EA is building vectors with leading ones, or vectors with leading zeros. The former means an convergence to fitness 1.0 whereas the latter means convergence to fitness 0.525. The reason for this is simple. At first the both leading ones and leading zeros behaves equally. The AI will then non-deterministically decide which of these will prefer. If zeros are chosen, it proceeds until reaching 21 leading zeros where the fitness will stop growing. In this state, the only way how to enlarge the fitness is to have 22 leading ones. However, the individuals are now too specialized on leading zeros and diversity for such a major switch is too low, so the AI gets stuck on fitness 0.525.

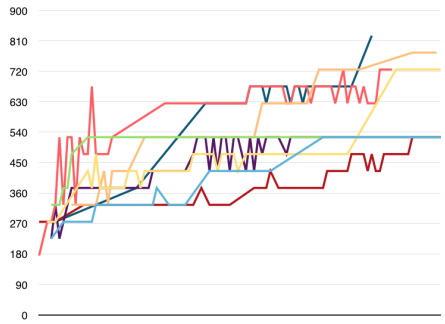


Figure 2:

c) Suprising Sequences

Encoding

Genotype is a bit vector with sequence symboles encoded in binary. Binary code of the symbols is aligned to the same size by adding leading zeros. Genotype-Phenotype conversion consist of grouping the bits into the string representing one symbol and binary-decimal conversion. Phenotype is therefore a list of symbols.

DIAGRAM

Fitness functions

Fitness functions for both globally and locally surprising sequences share the common procedure. This procedure checks if the sequence is surprising for given d (distance between symbols A and B). The procedure goes through the word, saves all the possible sub-words of the desired length and counts its duplicites.

For locally surprising is this procedure called just with *d=0*. In contrast, for globally surprising is this procedure called multiple times with d from range *<0, L - 2)*, where L is the length of the word.

Finally, the fitness is computed as *1/(1 + C)*, where C is the sum of all the duplicates found by the previous procedure.

Globally suprising sequences

S	Pop. size	Num. of gen.	L	Sequence
3	100	1	7	0, 0, 2, 1, 2, 0, 1
5	100	5	12	4, 1, 2, 0, 2, 3, 3, 0, 1, 4, 2, 1
10	100	5215	25	9, 2, 0, 3, 5, 6, 7, 8, 1, 4, 1, 7, 6, 2, 2, 9, 1, 8, 4, 0, 5, 8, 0, 6, 3
15	100	21	32	6, 9, 1, 14, 12, 14, 3, 5, 5, 2, 1, 12, 9, 7, 4, 8, 0, 7, 9, 10, 8, 11, 3, 6, 11, 13, 12, 7, 5, 14, 9,
20	100			

Locally suprising sequences

S	Pop. size	Num. of gen.	L	Sequence
3	100	3	10	0, 2, 1, 1, 0, 1, 2, 2, 0, 0
5	100	11	26	2, 0, 0, 1, 4, 1, 0, 3, 4, 0, 4, 3, 1, 1, 2, 1, 3, 3, 2, 4, 4, 2, 2, 3, 0, 2
10	100	2579	95	9, 5, 1, 8, 1, 5, 8, 6, 7, 5, 3, 0, 1, 0, 2, 1, 3, 6, 1, 7, 9, 1, 2, 9, 0, 6, 2, 4, 8, 9, 8, 0, 9, 4, 9, 7, 2, 2, 3, 1, 9, 3, 3, 2, 5, 0, 0, 4, 6, 3, 8, 4, 3, 9, 2, 8, 2, 7, 7, 4, 1, 4, 4, 0, 3, 4, 5, 6, 8, 7, 8, 3, 7, 6, 6, 4, 7, 3, 5, 4, 2, 0, 8, 5, 5, 7, 1, 6, 9, 9, 6, 0, 5, 2, 6
15	100	9258	195	6, 0, 3, 2, 6, 2, 9, 8, 5, 14, 4, 2, 10, 6, 7, 12, 10, 2, 11, 10, 8, 14, 8, 9, 13, 11, 6, 5, 3, 11, 8
20	100	7573	258	1, 7, 7, 11, 10, 9, 9, 4, 6, 14, 18, 5, 15, 8, 15, 6, 0, 0, 1, 13, 5, 8, 10, 5, 13, 17, 10, 0, 3, 12

d) Difficulty

- **One-Max** is the simplest of all. Fitness function is very straightforward and there are no problems with local maxima (the only maximum is the fitness 1.0).
- **Locally-surprising sequences**
- **Globally-surprising sequences**
- **LOLZ** is difficult problem mainly because of the existance of local maxima with leading zeros. From this state, it is very difficult for the AI to proceed and adapt to the new state with leading ones.