Project 2 IT3708

Programming an Evolutionary Algorithm

Iver Jordal

# Implementation

My solution is programmed in Python. I’ve used the concept of Object-Oriented Programming, for modularity. I will now explain what each class is responsible for.

**Main** - parses general arguments, initializes problem-specific code on demand, then a population is initialized with references to the problem-specific classes, and the EA loop is run. Main can perform many runs for the sake of averaging results. After the runs are performed, key statistics from all generations of all the runs are stored in a json file.

**Population** – Initializes random individuals and keeps them in two pools (children, adults). Also calculates and logs stats (f.ex. avg fitness) about the population.

**Individual** – Holds the phenotype and a reference to the genotype. This class can be extended so that its calculate\_phenotype method can be overridden easily.

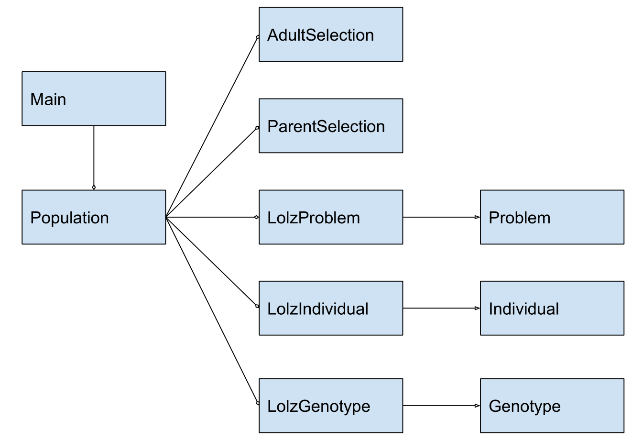
**Genotype** – Has a bit string and methods for mutation, crossover, cloning and random initialization. Also has a concept of age. This class can be extended so the methods can be overridden and data structures can be replaced.

**AdultSelection** – A class for dealing with *adult* selection. Upon initializing an instance of this class a specific adult selection method is specified, and that will become the selected adult selection method it will use. Generational mixing, over production and full generational replacement are implemented. In case other methods are needed, the class can be extended.

**ParentSelection** – A class for dealing with *parent* selection. Upon initializing an instance of this class a specific parent selection method is specified, and that will become the selected parent selection method it will use. Implemented selection methods: Fitness proportionate, Sigma scaling, Boltzmann selection and Tournament selection. If tournament selection is chosen, additional command line arguments (, ) are parsed.

**Problem** – A superclass that can be subclassed for specific problems. Subclasses must implement argument parsing and fitness evaluation.

## Modularity

****

*Figure 1: A simplified diagram that shows that Problem, Individual and Genotype are extended for the LOLZ problem.*

Here’s a code example that shows how Individual can be extended for the Surprising sequences problem:

**class** **SurprisingSequencesIndividual(**Individual**):**

**def** calculate\_phenotype**(**self**):**

self**.**phenotype **=** self**.**genotype**.**dna

**def** get\_phenotype\_repr**(**self**):**

**return** ', '**.**join**(**map**(**str**,** self**.**phenotype**))**

# The One-Max problem

## Full generational replacement, fitness-proportionate

Initial mutation rate and crossover rate is 50 %. With some trial and error, I arrived at a population size = 320, which yields a solution within 100 generations in approximately 98 - 100 % of the runs. If I lower it to 250, it gets the answer approximately 95 % of the runs.

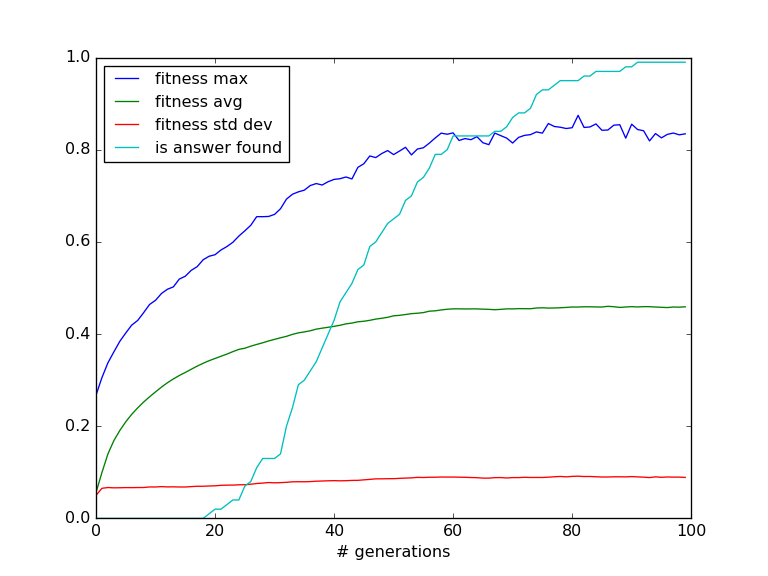


Figure 2: average stats with population size = 320

Next, I’ll try mutation rate 0.75 and then 0.25 without changing the crossover rate.

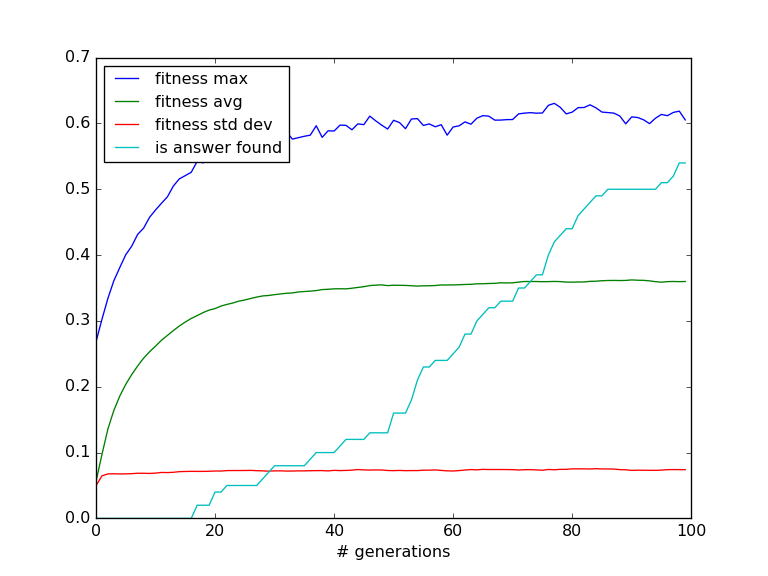


Figure 3: Mutation rate = 75 %, crossover rate = 50 %. Worse results than Figure 2.

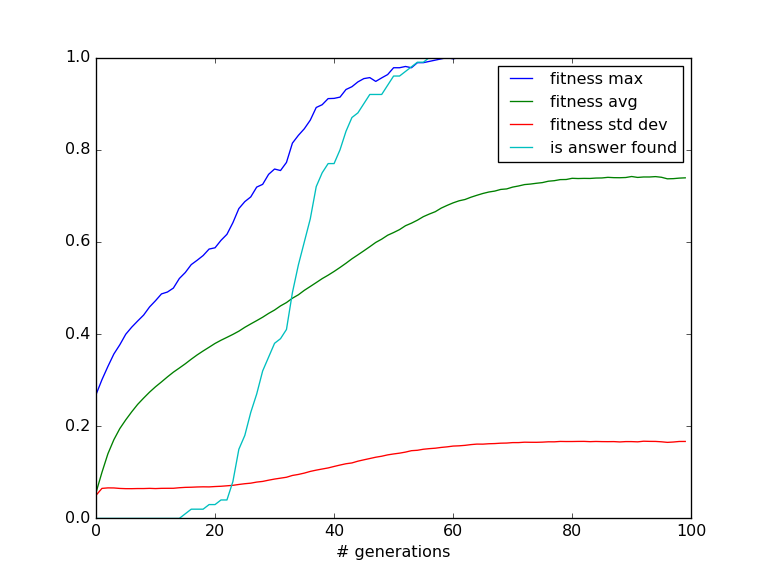


Figure 4: Mutation rate = 25 %, crossover rate = 50 %. Better results than Figure 2.

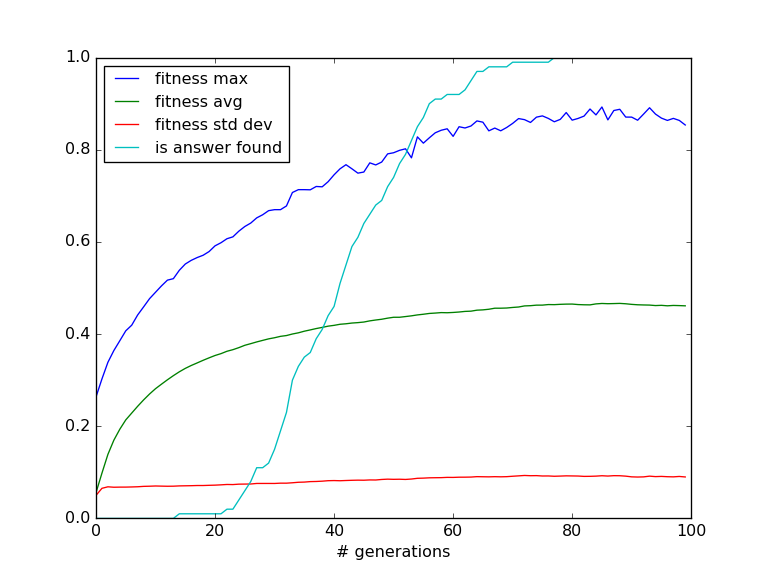


Figure 5: Mutation rate = 50 %, crossover rate = 75 %. Better results than Figure 2.

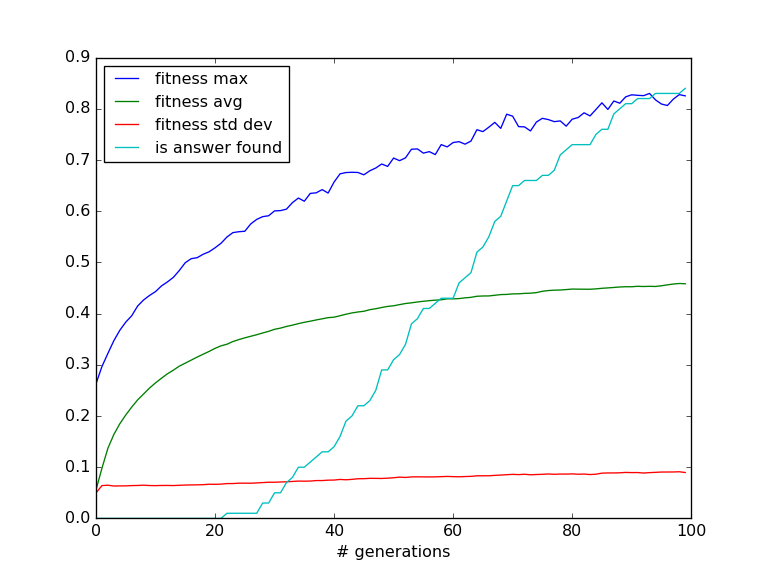


Figure 6: Mutation rate 50 %, crossover rate = 25 %. Worse results than Figure 2.

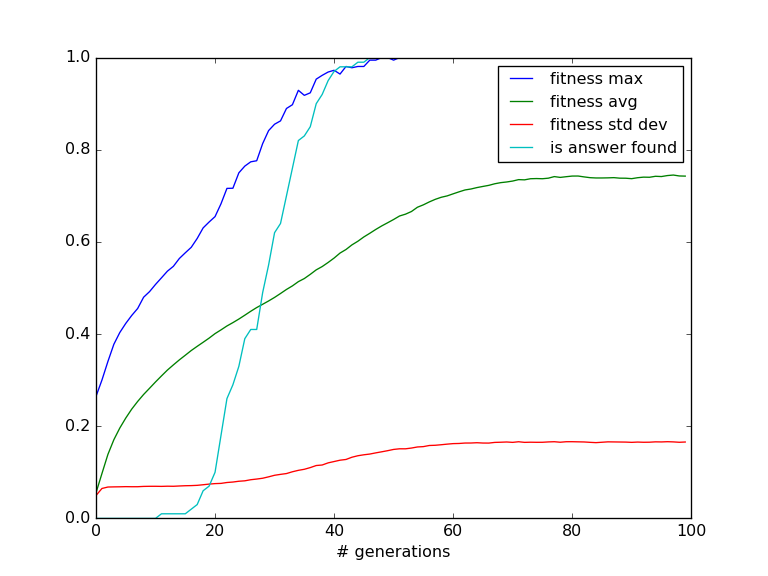


Figure 7: Mutation rate = 25 %, crossover rate = 75 %. Best result so far.

In conclusion, for the One-Max problem it is good to have a low mutation rate and a high crossover rate.

With the population size, mutation rate and crossover rate set in stone, I’ll experiment with the parent selection mechanism.

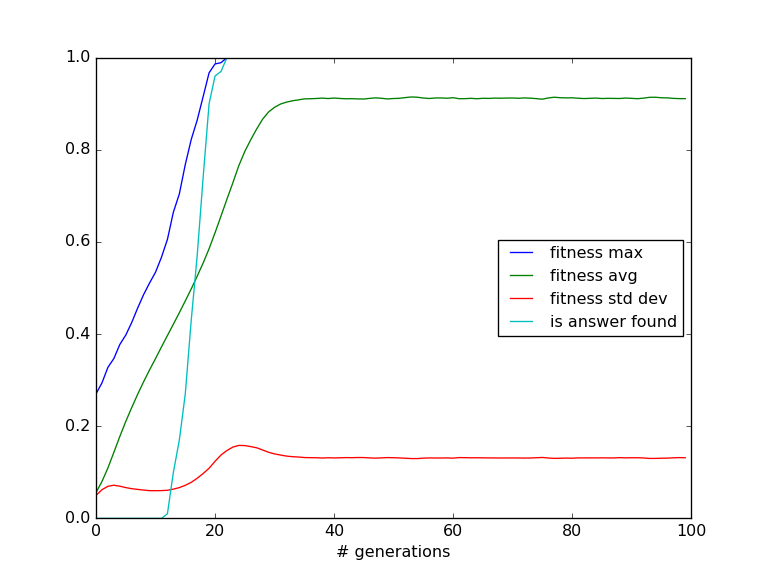


Figure 8: Using sigma scaling. Significantly improves results.

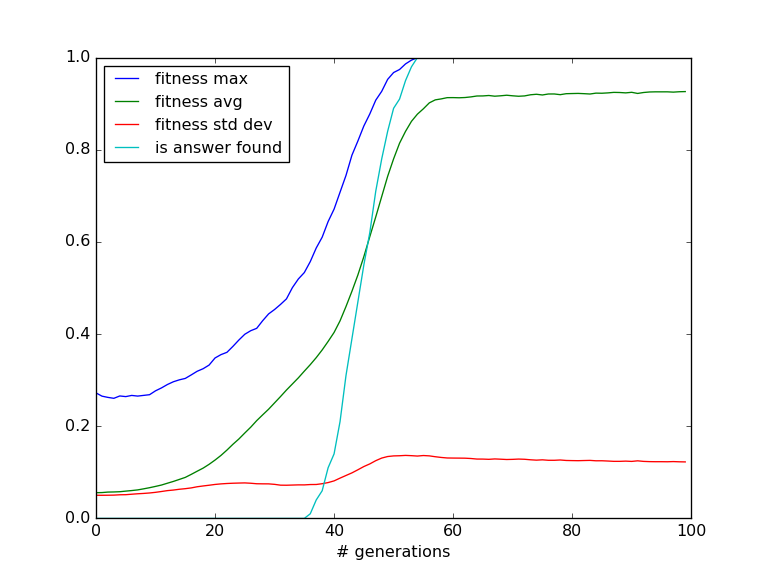


Figure 9: Using boltzmann selection. Yields slighly worse results than fitness proportionate.

Conclusively, sigma scaling is the best parent selection mechanism for this problem.

Next, the target bit string is modified to a random bit string instead of all ones. I do not expect this the increase the difficulty of the problem. Running with random bit string and otherwise same configuration as in Figure 8 confirms this, as Figure 8 and Figure 10 look almost identical.

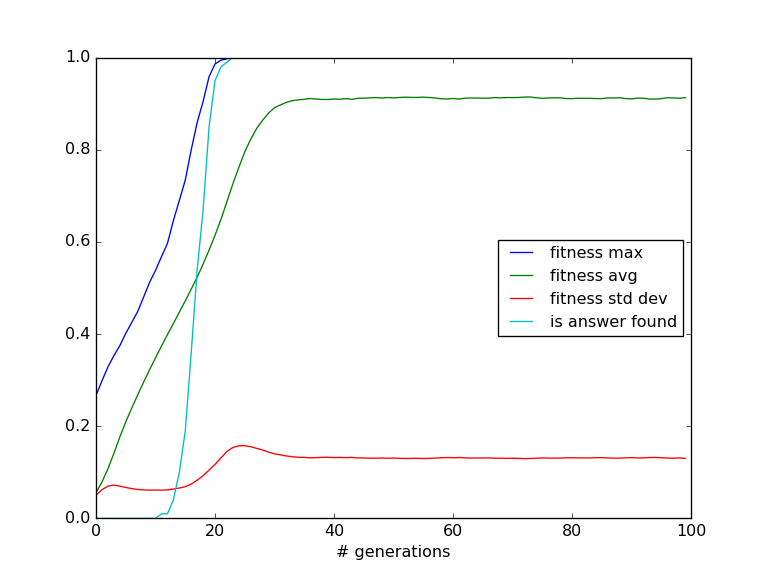


Figure 10: Using random target bit string instead of all ones. It does not affect problem difficulty.

# The LOLZ Prefix Problem

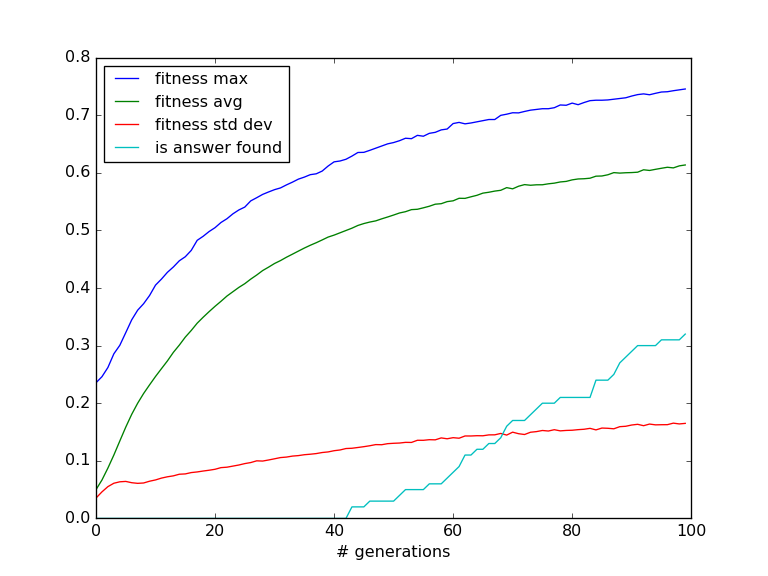


Figure 11: Stats from 100 runs of the LOLZ problem with 40 bits and z=21. It’s worth noting that only 32 of the 100 runs found an optimal solution within 100 generations.

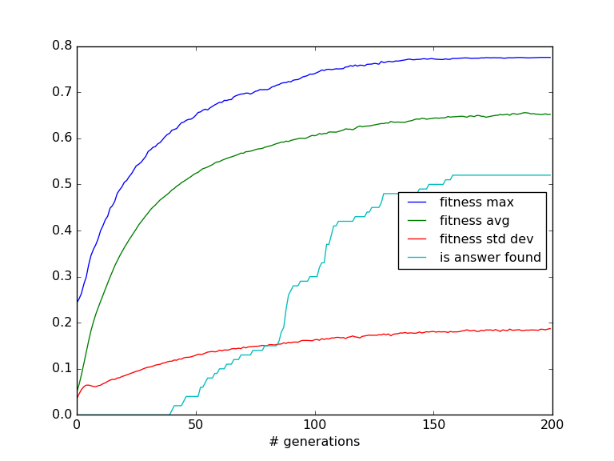


Figure 12: Same experiment as in Figure 11, but capped at 200 generations this time. The algorithm finds an answer in only 52 out of the 100 runs and the rest get stuck in local minima. In around half of the runs the algorithm finds zero prefixes to be good during evolution, but gets stuck when the score is capped at 21.

# Surprising sequences

## Genetic encoding

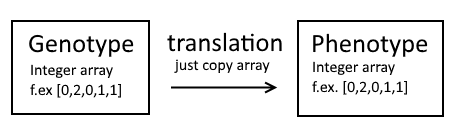


Figure 13: I chose a simple integer array representation for both the genotype and phenotype here, due to some problems with bit arrays when length \* (alphabet size) isn’t a power of two.

## Fitness function

The fitness function checks the number of repeating patterns. A pattern is repeating if there is more than one instance of (A, d, B). This is checked with a for loop within a for loop, where the outer loop iterates over start indexes and inner loop iterates over the distance, d. The fitness value becomes

Where is the number of repeating patterns, aka the number of *collisions*. This way, the fitness function yields 1 when a pattern is surprising, i.e. when it is a solution to the *globally* surprising sequences problem.

When checking for *locally* surprising patterns, the inner loop (for distance) is restricted to a single iteration, so that only the shortest distance d is considered.

# Difficulty

The One-Max problem is the easiest because the algorithm can be greedy without getting stuck in local minima. There's no need to strike a good balance between exploration and exploitation. This may be necessary for other problems, such as LOLZ, where the EA can end up with a non-optimal solution if the population is too small or the algorithm does not explore enough.