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 "pools" (children, adults, parents). 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 f.ex. 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 its methods can be overridden and the 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 (k, ϵ) are parsed.

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

Modularity

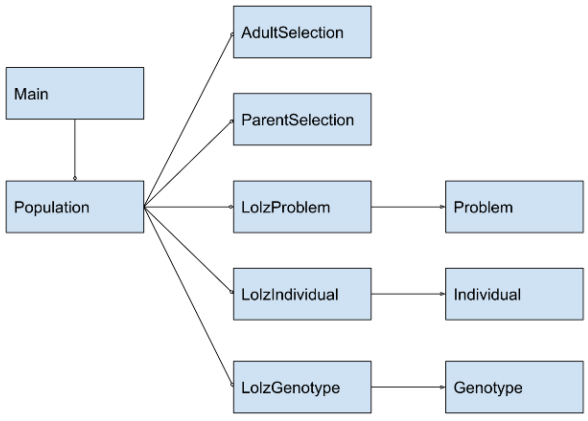

Figure 1: A simplified diagram that shows that Problem, Individual and Genotype are extended for the LOLZ problem. My solution actually has more classes, but they are omitted for brevity.

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 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.

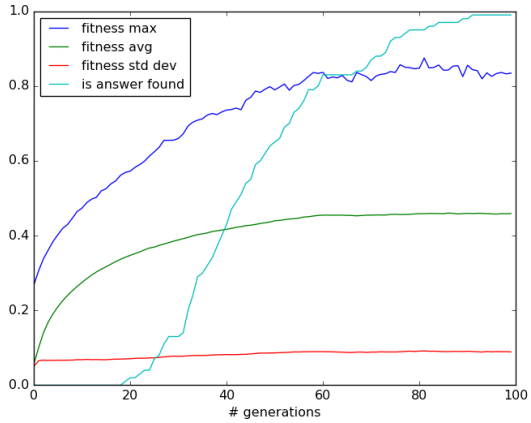

Figure 2: average stats with population size = 320. Notice the teal line that approaches 1 towards the end. This is what I looked for when tuning the population size. By the way, bear in mind that this graph and also the following graphs were constructed from average values across 100 runs.

Next, I'll experiment with mutation rate and crossover rate:

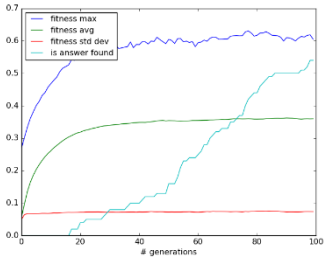

Figure 3: Mutation rate = 75 %, crossover rate = 50 %. Worse results than Figure 2.

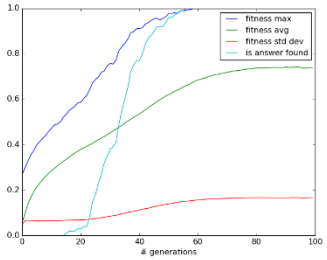

Figure 4: Mutation rate = 25 %, crossover rate = 50 %. Better results than Figure 2.

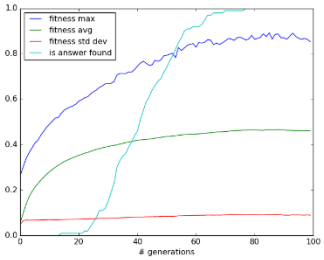

Figure 5: Mutation rate = 50 %, crossover rate = 75 %. Better results than Figure 2.

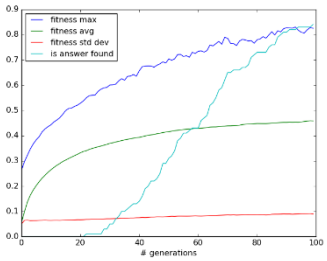

Figure 6: Mutation rate 50 %, crossover rate = 25 %. Worse results than Figure 2.

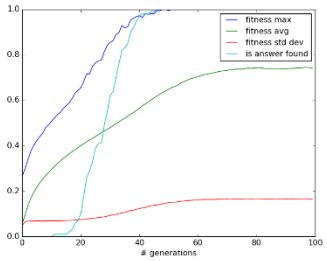

Figure 7: Mutation rate = 25 %, crossover rate = 75 %. Best result so far. Slightly better than Figure 4

Thus for this 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.

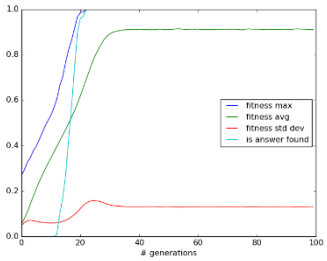

Figure 8: Using sigma scaling. Significantly improves results.

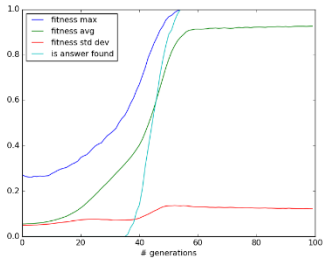

Figure 9: Using boltzmann selection. Slighly worse results than fitness proportionate.

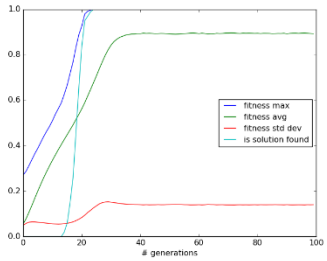

Figure 10: Using tournament selection. Almost as good as sigma scaling.

Conclusively, sigma scaling seems to be 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 before confirms this. The plots are almost identical. The plot is omitted for brevity.

The LOLZ Prefix Problem

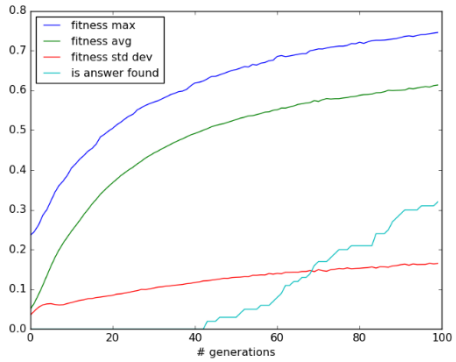

Figure 11: Stats from 100 runs of the LOLZ problem with 40 bits and z=21. It's worth noting that only 32 of the 100 runs found an optimal solution within 100 generations.

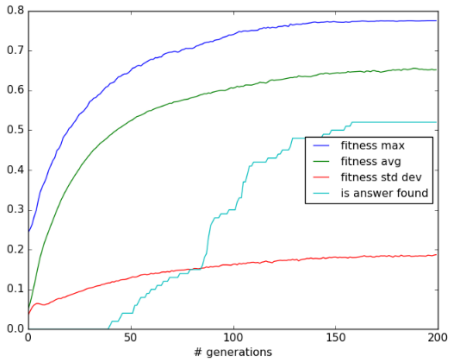

Figure 12: Same experiment as in Figure 11, but capped at 200 generations this time. The algorithm finds an answer in only 52 out of the 100 runs and the rest get stuck in local minima. In around half of the runs the algorithm finds zero prefixes to be good during evolution, but gets stuck when the raw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

Surprising sequences

Genetic encoding

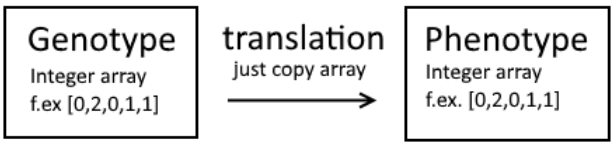

Figure 13: I chose a simple integer array representation for both the genotype and phenotype here, due to some problems with bit arrays when $length * (alphabet\ size)$ isn't a power of two.

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 $\frac{1}{1+e}$ where e 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 has no collisions and 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.

Locally surprising sequences

S	Pop' size	# gen	L	Sequence found
3	100	0	10	1, 0, 0, 1, 1, 2, 2, 0, 2, 1
5	100	5	26	4, 3, 3, 0, 3, 2, 3, 4, 1, 4, 0, 2, 1, 2, 4, 4, 2, 2, 0, 0, 1, 1, 3, 1, 0, 4
10	200	26	95	8, 3, 3, 1, 6, 5, 6, 8, 9, 1, 4, 3, 9, 5, 1, 1, 3, 4, 7, 0, 6, 4, 2, 3, 6, 9, 0, 9, 4, 8, 7, 9, 8, 1, 7, 8, 6, 2, 8, 4, 4, 5, 3, 5, 0, 3, 7, 6, 7, 5, 9, 9, 7, 7, 1, 0, 0, 4, 0, 5, 5, 2, 7, 4, 6, 1, 5, 4, 1, 9, 2, 5, 8, 8, 2, 0, 2, 2, 6, 3, 2, 1, 2, 4, 9, 6, 0, 7, 2, 9, 3, 8, 0, 8, 5
15	300	73	207	9, 3, 8, 4, 10, 4, 0, 10, 0, 11, 10, 9, 7, 12, 0, 1, 1, 9, 2, 3, 11, 13, 5, 12, 11, 14, 12, 12, 14, 10, 13, 10, 12, 8, 8, 2, 8, 0, 3, 10, 8, 13, 13, 14, 2, 14, 0, 7, 10, 14, 7, 0, 4, 5, 11, 12, 7, 13, 1, 6, 7, 4, 13, 7, 9, 4, 4, 14, 8, 1, 4, 1, 7, 3, 13, 8, 14, 14, 11, 7, 7, 8, 3, 0, 0, 13, 11, 2, 4, 3, 14, 3, 5, 14, 6, 1, 2, 2, 7, 5, 7, 14, 5, 13, 0, 5, 9, 5, 1, 0, 6, 11, 9, 10, 1, 10, 2, 5, 8, 10, 10, 11, 1, 13, 2, 1, 14, 9, 14, 4, 2, 10, 3, 1, 8, 6, 6, 10, 5, 0, 8, 9, 9, 1, 12, 4, 12, 3, 3, 6, 3, 12, 13, 9, 12, 5, 4, 11, 0, 12, 6, 4, 6, 5, 5, 3, 7, 6, 8, 11, 3, 4, 7, 11, 8, 7, 1, 5, 10, 7, 2, 6, 12, 2, 12, 10, 6, 13, 3, 2, 9, 0, 14, 1, 3, 9, 6, 2, 13, 12, 9, 13, 6, 9, 11, 4, 8
20	400	159	355	14, 14, 1, 9, 6, 0, 4, 18, 19, 18, 16, 7, 6, 11, 14, 11, 3, 2, 0, 16, 11, 16, 0, 6, 14, 17, 7, 19, 7, 8, 19, 14, 6, 10, 18, 14, 3, 3, 11, 8, 11, 18, 15, 14, 4, 1, 13, 14, 15, 8, 5, 9, 3, 16, 16, 9, 8, 6, 8, 10, 7, 13, 9, 9, 7, 9, 16, 10, 16, 5, 19, 6, 17, 3, 12, 4, 5, 7, 11, 11, 0, 5, 17, 19, 11, 17, 14, 16, 14, 19, 3, 18, 5, 15, 1, 3, 8, 17, 12, 6, 1, 11, 4, 2, 2, 8, 8, 4, 11, 5, 10, 19, 19, 10, 4, 17, 4, 7, 3, 10, 12, 16, 8, 9, 10, 9, 13, 8, 18, 12, 12, 1, 5, 13, 10, 5, 18, 4, 8, 13, 1, 8, 12, 17, 18, 11, 9, 17, 2, 17, 17, 16, 3, 7, 10, 17, 10, 6, 5, 3, 6, 18, 6, 12, 9, 4, 3, 15, 15, 4, 16,4, 15, 7, 7, 14, 8, 7, 2, 1, 2, 9, 19, 13, 13, 16, 6, 3, 17, 1, 18, 7, 17, 15, 11, 6, 9, 2, 5, 8, 16, 13, 3, 9, 0, 13, 7, 0, 7, 12, 10, 2, 15, 2, 19,15, 3, 19, 17, 6, 6, 19, 4, 9, 15, 9, 12, 14, 9, 11, 7, 16, 2, 14, 7, 1, 19, 8, 1, 12, 15, 6, 13, 19, 0, 18, 9, 5, 4, 14, 13, 5, 6, 16, 12, 19, 12, 0, 14, 0, 19, 16, 15, 13, 17, 8, 2, 13, 12, 5, 5, 1, 16, 1, 7, 4, 0, 0, 3, 1, 14, 12, 8, 0, 1, 15, 5, 11, 15, 19, 1, 17, 11, 1, 10, 15, 12, 2, 7, 18, 8, 3, 14, 18, 17, 13, 0, 2, 3, 4, 12, 3, 0, 17, 5, 16, 19, 2, 10, 10, 14, 5, 14, 2, 18, 0, 11, 12, 18, 1, 4, 4, 19, 9, 18, 10, 11, 13, 4, 10, 13, 11, 2, 4, 6, 7, 15, 0, 12, 11, 10, 1, 0, 10, 0

Globally surprising sequences

S	Pop' size	# gen	L	Sequence found
3	100	0	7	2, 0, 1, 2, 1, 0, 0
5	100	30	12	0, 3, 1, 4, 1, 0, 4, 2, 2, 0, 1, 3
10	200	34	25	4, 5, 1, 2, 3, 7, 0, 8, 2, 8, 4, 1, 6, 7, 5, 3, 6, 5, 9, 9, 7, 3, 2, 1, 0
15	300	17	37	6, 13, 12, 10, 1, 3, 9, 14, 0, 11, 2, 10, 7, 4, 8, 7, 2, 4, 5, 3, 5, 10, 11, 14, 1, 0, 6, 7, 13, 1, 14, 12, 12, 11, 9, 6, 8
20	400	35	50	1, 16, 10, 11, 3, 18, 19, 15, 8, 14, 9, 17, 8, 19, 14, 2, 18, 5, 7, 6, 1, 17, 4, 13, 0, 10, 12, 9, 11, 12, 0, 6, 6, 10, 1, 15, 3, 17, 13, 5, 16, 8, 16, 18, 15, 9, 4, 11, 7, 19

Difficulty

The One-Max problem is the easiest problem because the algorithm can be greedy/naive without getting stuck in local minima. Every increase in fitness gets you one step closer to the ultimate solution. 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. Then there is the more computationally expensive problem of surprising sequences. I'd say this is the hardest (of the problems in this project) to solve for an EA. For example, the max length of a locally surprising sequence is $S^2 + 1$, but my EA algorithm didn't even come close to that number for $S = 20$. The larger the S, the harder the problem is to solve. There are many possible solutions and also a very large number of nonsolutions. I think globally surprising sequences are harder than locally surprising sequences because 1) the fitness calculation is more complex (more distances to check), 2) every integer in the array plays a role in a broader context, and one small change can have negative consequences (collisions) in unexpected places and 3) given a phenotype with a fitness that is almost 1, it is not guaranteed that there exists a small mutation that will up the fitness to 1. That said, globally surprising sequences are generally shorter than locally surprising sequences (because the constraints are tougher), and shorter sequences means less data to mutate.