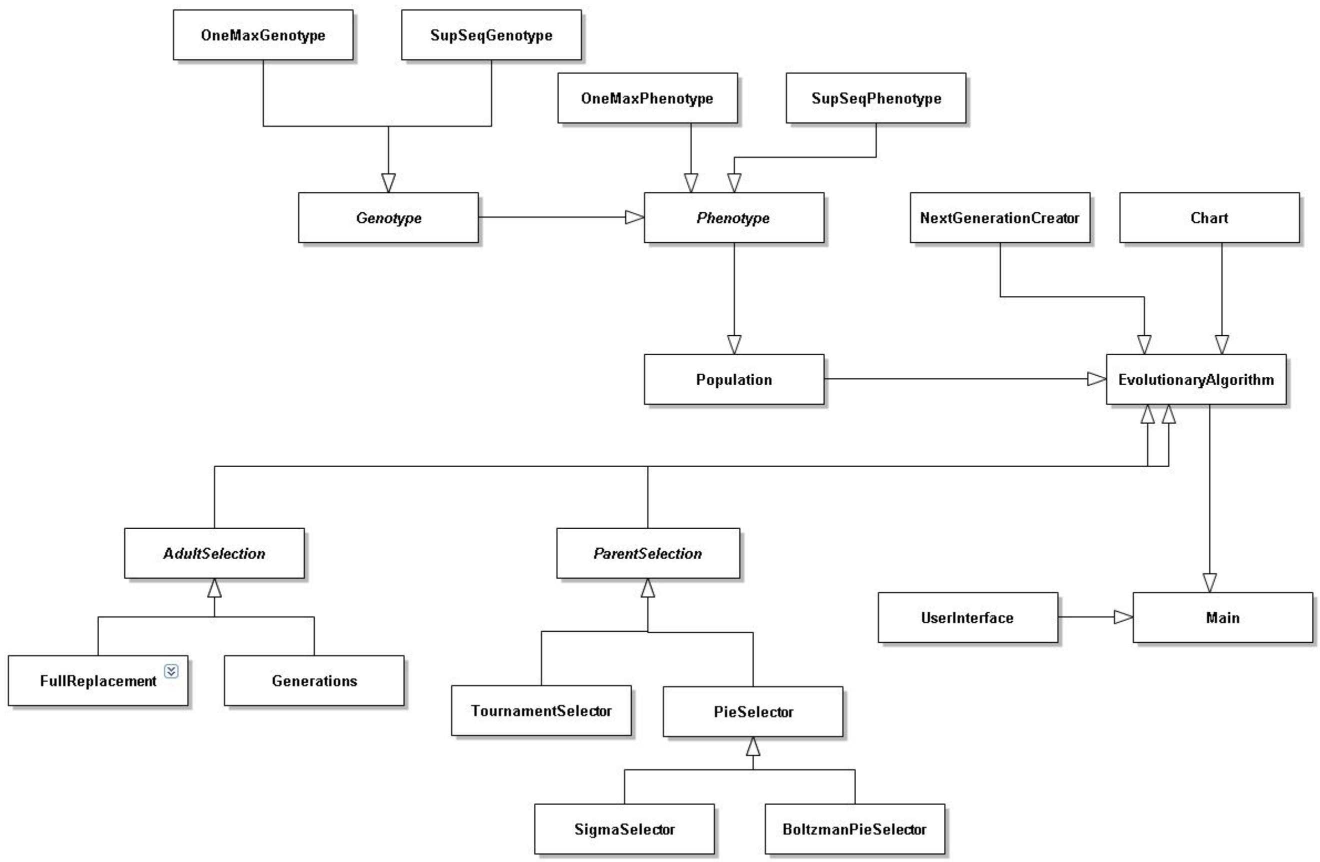
Programming the Basics of an Evolutionary Algorithm(EA)

By Kristoffer Hagen

1. The algorithm

To create the evolutionary algorithm I decided to use Java with the addition of a library to help me produce the fitness plots. I wanted separate classes for most of the different aspects of the algorithm in order to make it very modular. Here is the class diagram for the program along with a brief description of the classes:



Genotype: contains attributes and methods for genotypes and is inherited by the problem-specific genotypes.
Phenotype: created from the genotype class. Contains methods for generating fitness and mating of two phenotypes that creates a new genotype.
Population: a list of phenotypes and contain methods to gain useful information about the population.
AdultSelectors: takes in two populations (the previous adults and the new chrildren) and returns the next generation. These classes have a superclass and several different subclasses.
ParentSelectors: takes in a population and determines who gets to be parents. Returns a list of Phenotypes. Like AdultSelector has a superclass and several different subclasses.
EA: Contains the main loop and stores all the inputs needed

2. The Modularity

As can be seen in the class diagram and the description of the classes, I tried to make it as modular as possible. Implementing a new adult selector or a parent selector just inherits from the base class and specifies the new desired behavior. The same can be said for genotype and phenotype. To create a new problem, the only programming that needs to be done is to inherit the Genotype and Phenotype classes, write the inherited functions the are problem specific, such as fitness generation, data representation and set some crossover variables depending on the data representation.

All the rest of the code is independent of the Genotypes and Phenotypes used. All the Adult- and ParentSelectors work with any new problems, the same goes for the main EA loop.

```
public EvolutionaryAlgorithm(int popSize, AdultSelector as, ParentSelector ps, int steps, createNextGen cg, Genotype g, int targetfit){
    this.lastGen = initPopulation(popSize, g);
    this.nextGen = initPopulation(popSize, g);
    ArrayList<Genotype> parents = new ArrayList<Genotype>();

    XYChart chart = new XYChart("Fitness Chart");

    this.adultSelector = as;
    this.parentSelector = ps;
    this.cg = cg;

    int i = 0;
    while(i<steps && lastGen.getBest()<targetfit){
        i++;
        System.out.println("Generation "+i);
        chart.getPopData(lastGen, i);

        lastGen = as.generateNextPopulation(lastGen, nextGen);
        chart.getPopData(lastGen, i);
        parents = ps.generateNextGeneration(lastGen,((1.0*i)/(1.0*steps)));
        nextGen = cg.createChrildren(parents);
    }
```

Here my EA loop can be seen together with the input needed to run it. Here an example on how to use it

```
= new EvolutionaryAlgorithm(500 , new FullReplacement(1, true), new TournamentSelector(50,.8),
                            100, new createNextGen(1 , .05 , 3 ),new OneMaxGenotype(40) ,40);
```

This will create a new EA with: 500 population, Full generational replacement with elitism, Tournament selection with K=50 and epsilon=0.8, 100% crossover, mutation rate = 0.05, every allele comes from a random of the two parents, onemax problem with 40 size, target fitness is 40.

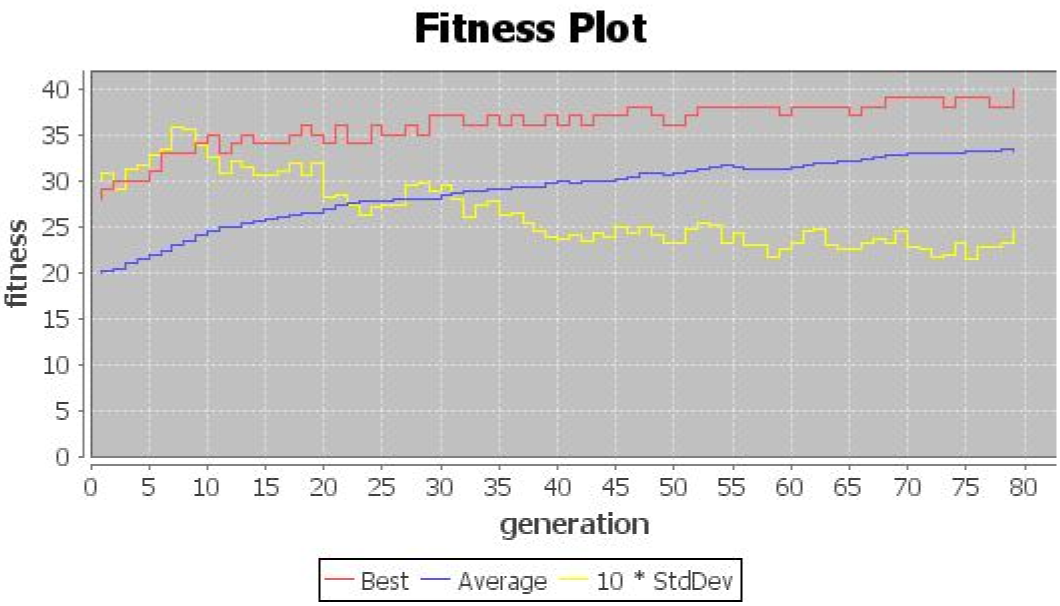
```
new EvolutionaryAlgorithm(500 , new FullReplacement(1, true), new TournamentSelector(50,.8),
                          100, new createNextGen(1 , .05 , 3 ),new SuprisingGenotype(10, 25) ,0);
```

This will do the same but for the Surprising Sequence problem with 10 symbols and 25 for length, with target fitness 0. (I do negative fitness for this one).

3-4. The Testing

Starting out with full generational replacement without elitism (elitism simply stores the best member from last generation) and the fitness-proportionate parent selector (from now on called PieSelector). After a few tries I found that with a population of 500 I consistently solved it before 100 generations.

Fitness plot of that run:



Then I started testing to find the best values and choices for the different parameters. I ran every test 5 times and took the average of the 5 to even out the randomness. I had 3 adults electors, 4 parent selectors and 3 crossover methods to try, together with mutation rate. NA means no solution was found within 100 generations. The table the following page covers all the test. First I tested my three different crossover methods, assigning every allele, or bit, to inherit from a random of the two parents had the best result (though this is very specific to this problem).

Next I tested different mutation rates, a mutation will flip one bit at a random position. I got the best result with a low mutation rate.

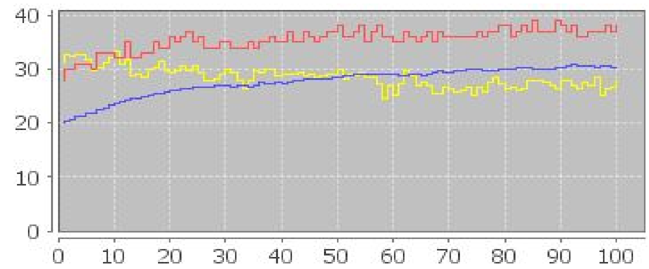
Finally I tested all my adult- and parent selectors. Surprisingly to me it turned out the the tournament selection had the best results out of the parent selectors and the generational among the adult selectors. My Generational selectors puts all the parents and the children in the same pool and picks out, with some randomness, the best of them. I am rather sure that this would be very different on other problems, as the onemax problem values exploitation very much and does not have any local maxima. This leads to the greedies selection being the most successful.

	Population	Adult Selection	Parent Selection	Mutation Rate	Crossover method	Generations for solution:
1	500	Full replacement no elitism	PieSelector (Proportional)	0.2	Split random	71
2	500	Full replacement no elitism	PieSelector (Proportional)	0.2	Split middle	NA
3	500	Full replacement no elitism	PieSelector (Proportional)	0.2	**Random every allele**	62
4	500	Full replacement no elitism	PieSelector (Proportional)	0.1	Split random	75
5	500	Full replacement no elitism	PieSelector (Proportional)	0.5	Split random	NA
6	500	Full replacement no elitism	PieSelector (Proportional)	0.05	Split random	60
7	500	Full replacement no elitism	PieSelector (Proportional)	0.1	Random every allele	50
8	500	Full replacement no elitism	PieSelector (Proportional)	0.5	Random every allele	NA
9	500	Full replacement no elitism	PieSelector (Proportional)	**0.05**	Random every allele	49
10	500	Full replacement with elitism	PieSelector (Proportional)	0.05	Random every allele	41
11	500	**Generational**	PieSelector (Proportional)	0.05	Random every allele	12
12	500	Full replacement no elitism	SigmaPieSelector	0.05	Random every allele	NA
13	500	Full replacement with elitism	SigmaPieSelector	0.05	Random every allele	82
14	500	Generational	SigmaPieSelector	0.05	Random every allele	11
15	500	Full replacement no elitism	BoltzmanPieSelector	0.05	Random every allele	17
16	500	Full replacement with elitism	BoltzmanPieSelector	0.05	Random every allele	16
17	500	Generational	BoltzmanPieSelector	0.05	Random every allele	11
18	500	Full replacement no elitism	**Tournament(K=50, e=0.9)**	0.05	Random every allele	5
19	500	Full replacement with elitism	Tournament(K=50, e=0.9)	0.05	Random every allele	5
20	500	Generational	Tournament(K=50, e=0.9)	0.05	Random every allele	4
21	500	Generational	Tournament(K=50, e=0.9)	0.05	Random every allele	4

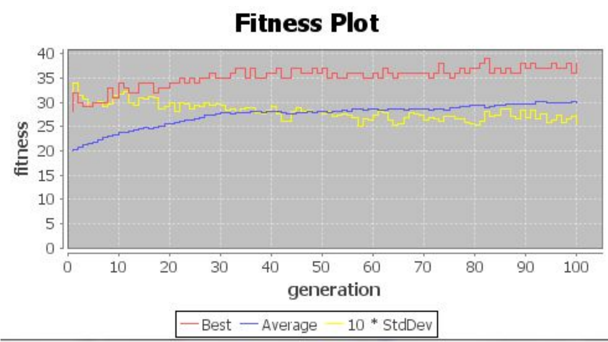
It is not possible to show all the fitness plots here without exceeding 10 pages so I will select a few to show different interesting differences. The different numbers refer to the numbers left on the table above.

4: low mutation, solution

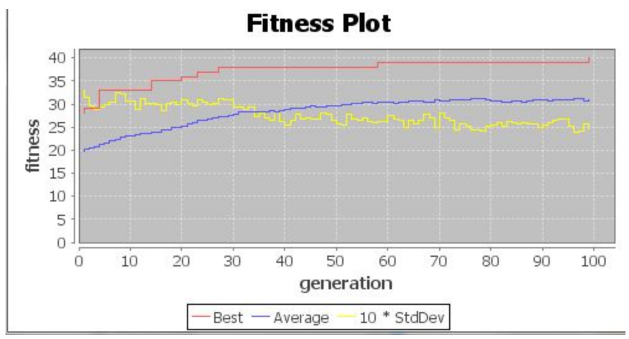
5: high mutation, no solution



12: no elitism, no solution

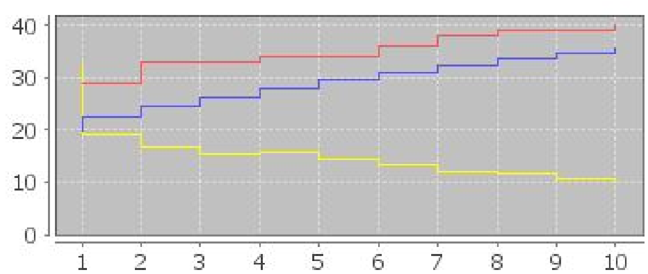


13: elitism and solution

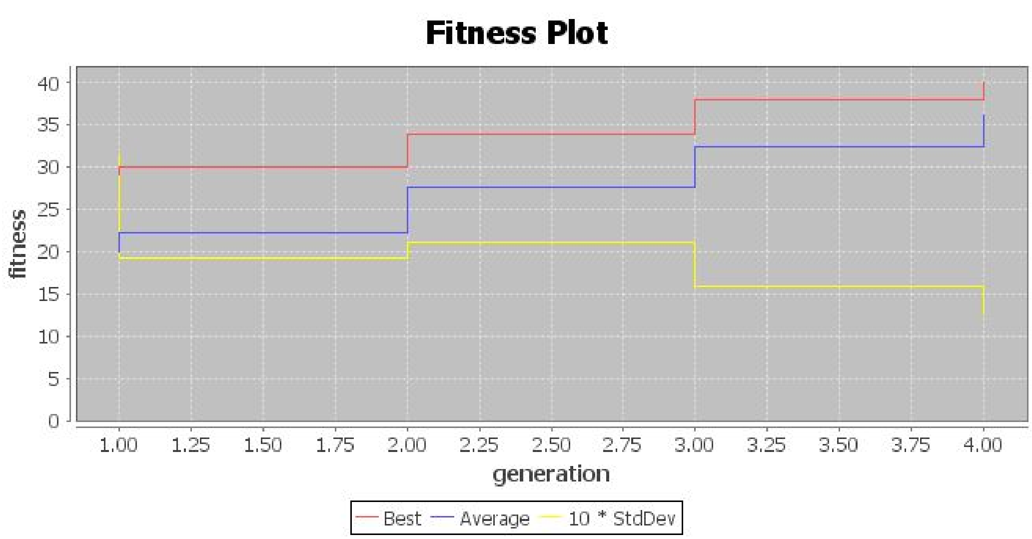


9: full replace no elitism

11: generational



21: best setup for this problem:



5 Change target string

Changing the target string to, instead of being all 1's is now alternating 1 and 0s (ie. 10101010…..

This had no effect on the results whatsoever. Comparing the results to four different runs in the table (still using the average of 5 runs):

Run	11111111…..	1010101010…..
1 in table	71	92+37+58+96+64= 70
7 in table	50	60+52+50+46+57= 53
13 in table	82	92+96+69+92+66= 83
21 in table	4	5+5+4+4+5= 5

I expected this to have no effect on the results at all. The algorithm doesn't care what the target bit is, and there is no functional difference between the 1111.. and the 101010… string. The target string could have been ANY 40 bit long string and the algorithm would perform the same.

Lastly, sorry I made this too long, the text really doesn't even fill 2 pages, but without exceeding the 4 pages desired I would have to make the images and graph too small to read and you would hurt your eyes. :(