Programming the Basics of an Evolutionary Algorithm (EA)

The One-Max Problem

The code for the project is written in C++. To generate the plots you need to have matlab installed.

1. Algorithm Description

Class diagram

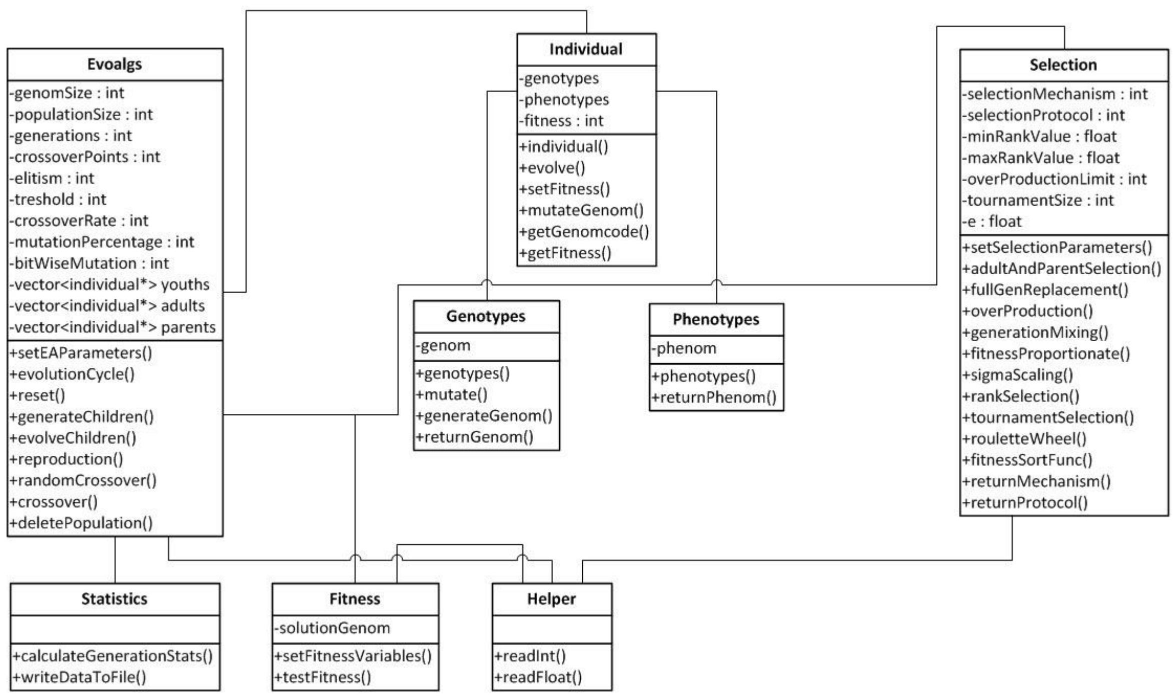


Figure 1: Class diagram made in Microsoft Visio

As shown in figure 1 we have a total of 8 classes. Evoalgs is the main algorithm class which controls the evolutionary loop and holds the current population, and the main algorithm parameters. The evolutionary loop starts with an initial population of children, which are then evolved into juveniles. We then run a fitness test on the entire population, and write information about the current generation to file, which are later plotted to illustrate the change from generation to generation. Adults are then selected both from the youth and current adult pool based on the selection protocol, before the most suitable parents are selected based on the selected selection mechanism. We then use these parents to generate a new generation of children by means of genetic crossover and mutation. These children are then used as the starting point of our new evolutionary cycle. We chose to split our population into three vectors: Youths, adults and parents. Where youths are the new generation, adults are the retained adults and parents are the adults which are selected for reproduction.

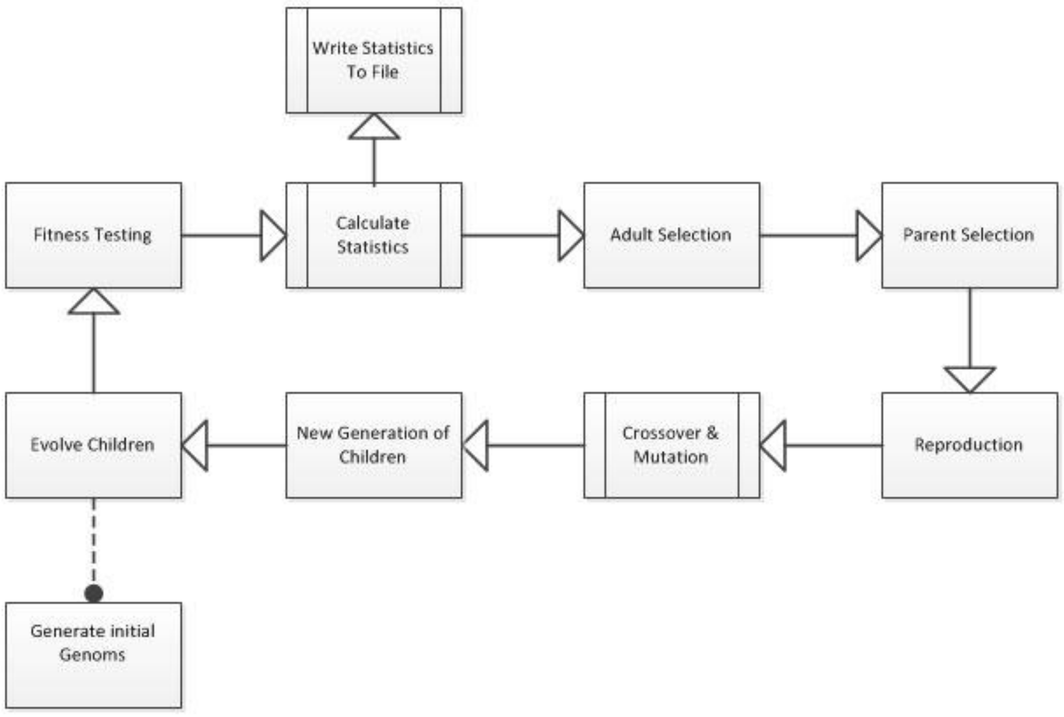


Figure 2: Illustration of the Evolutionary Cycle made in Microsoft Visio

Our fitness class is responsible for evaluating the fitness of the current generation and assigning fitness values to all our individuals. Our population consists of individuals, which contains a genotype, phenotype and a fitness value. The genotype class represents a genom and contains a genocode and related functions such as mutation. The phenotypes class contains a phenocode.

The selection class contains the selection variables and the adult and parent selection functions. We have implemented full generational replacement, over production generational mixing protocols for adult selection. The current version includes fitness-proportionate, sigma scaling, rank selection and tournament selection mechanisms for parent selection. We have also included elitism as a choice, where you can choose to keep the N fittest individuals from a generation and pass these untouched to the next generation.

The helper class contains various helper functions which can be reused in multiple places throughout our code. The statistics class contains functions for calculating statistics and writing run data to file. This later can be used to help us determine the best parameters.

2. Modularity and Reusability

Since the phenotype will vary from problem to problem you will have to write a new mapping function from genotype to phenotype for each problem. This means that you will have to rewrite only the phenotype constructor which takes a genocode as input. If you wish to change the genotype function you most likely only have to rewrite the constructor here as well.

Each problem also requires a new fitness evaluation function. The fitness function takes all juveniles in the current generation as a parameter and you therefor only need to change the function itself.

To add new genetic operators such as a new crossover function, you can easily write a new function or modify the current function and use this in our reproduction function.

```
genotypes::genotypes(int genomSize)
phenotypes::phenotypes(vector<int> genomCode)
void fitness::testFitness(vector<individual*> generation)
```

Code sample 1: Function headers for the genotype constructor, phenotype constructor and our fitness testing function

The statistics class is also easily expandable if you wish to add more functionality.

To incorporate new selection mechanisms or protocols you only need to add the additional variables and functions needed in the selection class. And make it selectable for the user. You also have to add another else if in the adult and parent selection function. All in all you might have to write 5 lines of new code in addition to the function itself.

```
if(selectionMechanism == FITNESS){              //Fitness proportionate
       fitnessProportionate(adults, parents, count);
}
else if(selectionMechanism == SIGMA){           //Sigma Scaling
       sigmaScaling(adults, parents, count);
}
else if(selectionMechanism == RANK){            //Rank Selection
       rankSelection(adults, parents, count);
}
else {  //selectionMechanism == TOURNAMENT     //Tournament selection
       tournamentSelection(adults, parents, count, elitism);
}
```

Code sample 2: Parent Selection

Our only weakness that we see when it comes to reusability is that we might have to redefine some data types since we have not used typedefinitions. But the implications of this are minimal, so we chose to ignore it.

3. Performance Analysis 40-bit One-Max Problem

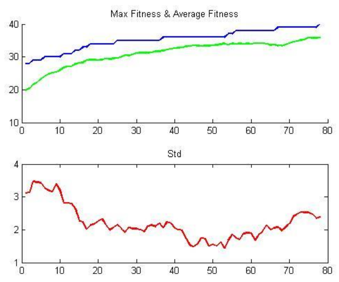
When starting to test out the performance of our algorithm we struggled a bit finding a solution using full generational replacement and fitness proportionate. We therefore chose to implement elitism to be able to find a solution using smaller population sizes.

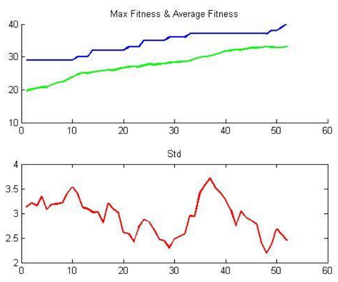
We started at an arbitrary chosen population size of 500, elitism set to one, crossover rate set to 0.2, mutation rate to 0.2 and a 1 bit mutation. With the same settings we could go as low as a population size of 150 before we no longer could find a solution consistently, where the solution on average was found after 70-80 generations.

When setting elitism to 2 we could go as low as a population size of 80 and still find the solution consistently in about 80 generations on average after tweaking the variables for a while. We seemed to get the best results with crossover set to 0.2, mutation rate set to 0.15 and the bit wise mutation rate to 1 bit.

When elitism is set to one we can weren't able to get the population size lower than about 170 with the same settings to more or less constantly find a solution when trying to tweak the parameters.

What we experienced when experimenting with different settings is that a high mutation rate causes the average fitness to converge much faster at a lower value than when using a lower mutation rate. This is because most bits are set to one after a while running the program and mutations are therefor most likely to flip a bit set to one. Using a higher mutation rate and a higher bitwise mutation caused the algorithm to struggle a lot more in finding a solution.





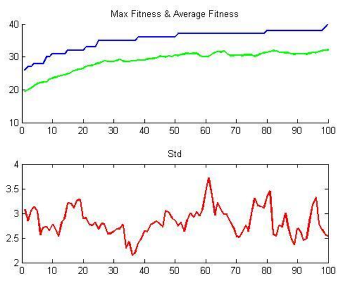


Figure 3, 4 and 5: Figure 3 shows the plot for a population size of 90 have elitism set to 2. Figure 4 shows a population size of 150 having elitism set to 1. Figure 5 shows a population size set to 300, mutation rate set to 0.5 and 3 bit mutation.

We also tested out different crossover functions. One function splits both genomes into N equally sized segments and swaps them. The other function uses one randomly selected crossover point. What we found is that the crossover function with the randomly selected crossover point works a lot better than the one with fixed crossover points. After running multiple simulations we found that the fixed point crossover function requires a population size of about 300 to consistently find a solution, while the crossover function with randomly selected crossover points could do the same with roughly half the population size. The main reason for this is that many genotypes quickly becomes too similar and converges on a local maximum when using a fixed crossover point(s).

Our conclusion is that using a higher value on elitism greatly increases the performance when using full generational replacement and fitness proportionate parent selection. A crossover rate between 0.15-0.25, a mutation rate between 0.10-0.20 and a bitwise mutation of one bit seemed to yield the best results in our runs.

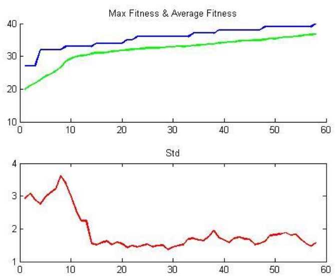
4. Experimenting with Parent Selection Mechanisms

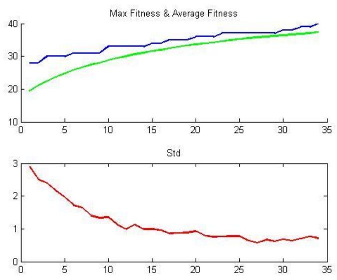
For the following tests we used a population size of 150, elitism set to one, a crossover rate of 0.2 with the random crossover point function and the mutation rate set to 0.15.

We started out with sigma scaling which gave us a lot better results than with fitness proportionate. It consistently found the solution between 35 and 80 generations at an average of about 60.

Rank selection gave consistently found the solution in between 30 and 90 generations with an average at about 55 with min set to 0.5 and max set to 1.5. When changing the min value to 0.0 and 2.0 we didn't notice much difference and found a solution in the same about of generations. The only noticeable difference was a more varying standard deviation with a higher selection pressure.

When testing out tournament selection we used an e value of 0.7 and tested out three different tournament sizes. When using a tournament size of 3 we found the solution on average in about 35 generations. And when increasing the tournament this number of generations went down. When using a tournament size of 10 we found a solution on ~20 generations on average.





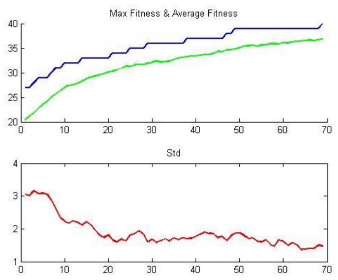
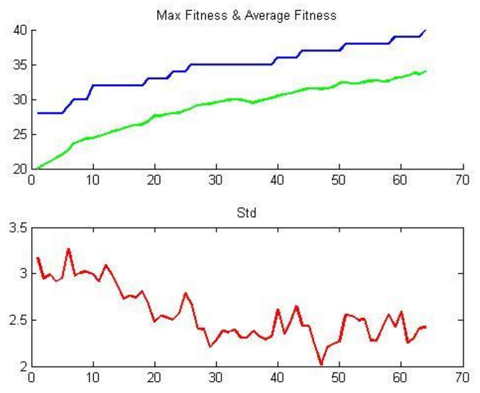


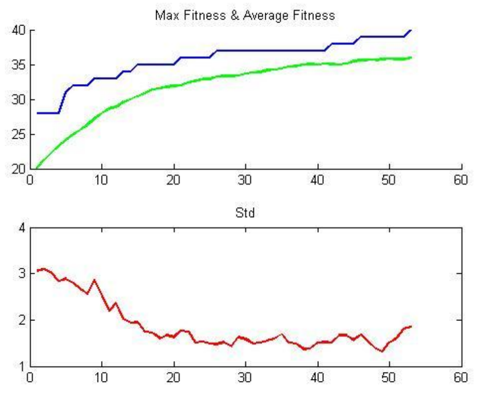
Figure 6, 7 and 8: Figure 6 shows the plot for sigma selection, Figure 7 shows the plot for tournament selection with tournament size set to 3 and e to 0.7, Figure 8 shows the plot for rank selection with min set to 0.5 and max to 1.5

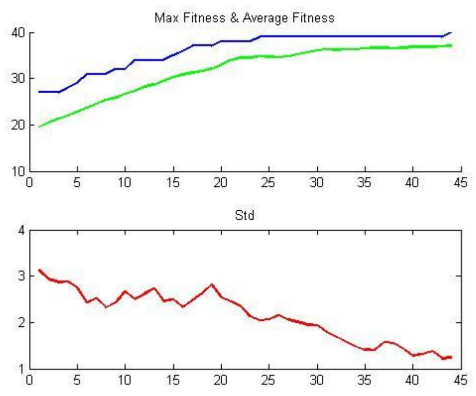
Looking at the plots you can see that tournament selection is the most powerful selection mechanism by far. But we're not convinced that tournament selection normally should outperform the other two since sigma scaling and rank selection normally should perform well in these kinds of problems where the fitness fast becomes very homogenous.

5. Random Bit vector Solution

One might think that having a random bit vector as a solution one might be so lucky to generate a genome in the initial solution that is closer to the solution than one would having a target bit vector only consisting of ones. But in theory there should be no difference, since you still only have a 50% chance of generating the right bit. To see if I could spot any difference I tested out all four different selection mechanisms with the same settings as in part 4.







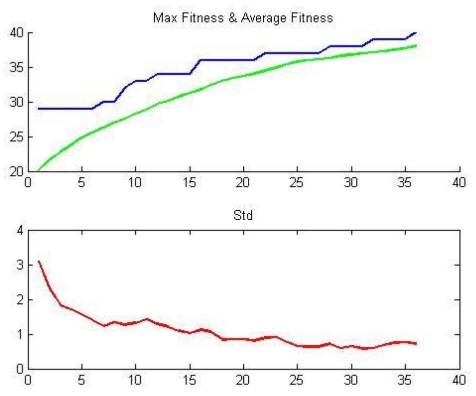


Figure 9, 10, 11, 12: Figure 9 Shows a plot of fitness proportionate, Figure 10 Shows a plot of rank selection, Figure 11 shows a plot of Sigma selection and Figure 12 shows a plot of tournament selection with a tournament size of 3.

Looking at the plots it's hard to spot any obvious differences. Looking at the number of generations it takes to find a solution you can see that it's roughly the same, but using the number of generations to find a solution based on two individuals runs is not a good measure of similarity. The only thing I can conclude with is that the algorithm didn't seem to struggle more finding a solution with a random bit vector target compared to a solution consisting only of ones and could still find the solution every run in roughly the same amount of generations.