Project 1

Håkon Gimse, Subsymbolic Methods in AI

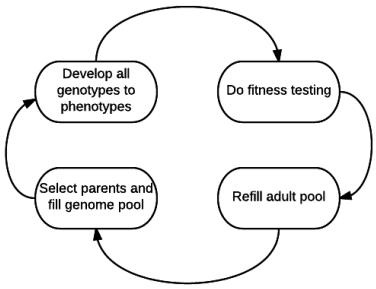
Document your implementation
My implementation is started by runing the GUI class. After selecting parameters in the interface and pressing the start button it instantiates a EvolutionaryAlgorithm instance using those parameters. This is the class in charge of running the steps in the EA life cycle. It has a population of solutions as several pools (lists) of genotypes and phenotypes. One pool for genotypes, one for phenotypes before they mature into adults/parents and one pool for adult phenotypes. These are potential parents, and can make new genomes. I have a generic class called Genotype. This class has one important variable, the dna vector. It also has a limit of how many configurations a dna component can have, i.e. 2 in the OneMax problem. I have implemented the genetic operators on genotype level, so it has a method for mutation and for crossover. My solution supports either component mutation or individual mutation.

I have a base class for Phenotypes, and every problem has a spesific phenotype class, that inherits the base class Phenotype. This is to ensure that every problem spesific Phenotype implementation at least has methods for developing from genotypes and evaluating the phenotypes fitness. Genotypes develop into phenotypes. This is done by instantiating a Phenotype class instance using the genotype as a parameter. I have a development method in the initialization function of the phenotype. This function translates the dna from the genome into the phenotypes respective dna.

EA Code
The EA life cycle is shown below. The genotype-phenotype development phase is described above. The fitness testing is dependant on the problem, but I always scale the fitness between 0 and 1, where 1 is the optimal solution.The surprising sequence calculation is shown below. The next step is to refill the adult pool. This is dependant on the adult selection protocol selected. It is done by slicing a sorted pool list. When mixing is used i sort the concatination of the adult and the child pool and replace the adult pool with the best of this concatination.

The last step is to select the parents and fill genome pool. The concept is to mate two and two parents and each pair resulting in two new genotypes. How a parents is selected is also dependant on the protocol selected. If the protocol is Fitness proportionate, Sigma-scaling of Boltzman selection I run through the entire adult pool and scale each fitness value into a piece of a cake, totaling to 1. After this is done, I just use the basic roulette selection described in the assignment. If the selection protocol is tournament, I create two tournaments for each parent pair. A tournament is a pool of randomly selected adults. This pool does not share any adults with the other tournament pool. The champion and parent of a pool is either the one with the best scoring fitness or a random other choice, dependant on the random “slip-through” value.

When two parents are found they mix their dna and create a new dna vector. This is done using a *crossover* function. I have implemented both random and fixed points of crossover, but the concept is that the parents dna is mixed. Performance wise, the random crossover point method performs much better. The resulting dna vector is pased to a Genotype constructor and becomes the child.



Reusability and modularity
The Genotype class should be the only genotype you need, as the genetic operators are written in a way that allows any range of integers. The following code snippit is an example showing that mutation happens randomy and efficently, and a component mutates into something else. It does not matter whether a component can be [0, 1] or [0, 1000].

```
if mutation_protocol == 1:
    #  Invividual
    r = random()
    if r <= mutation_rate:
        r_i = randrange(len(self.dna_vector))
        old_val = self.dna_vector[r_i]
        new_val = randrange(self.symbol_set_size - 1)
        if new_val >= old_val:
            new_val += 1
        self.dna_vector[r_i] = new_val
```

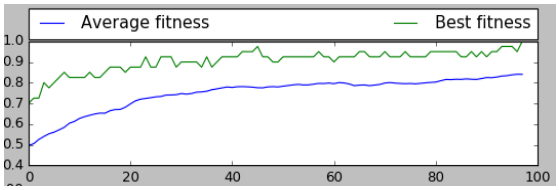
If I want a representation to allow doubles, I can just represent it as integers and then translate it in the development method. The crossover operator is also completely independant of the problem as shown below.

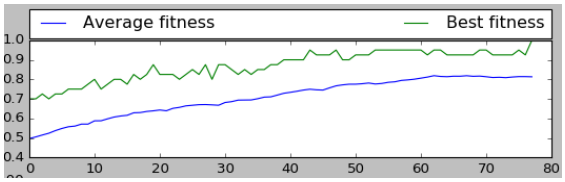
```
for i in range(len(crossover_points)):
    point = crossover_points[i]
    if i % 2 == 0:
        child1_dna_vector += copy(self.dna_vector[last_point:point])
        child2_dna_vector += copy(other.dna_vector[last_point:point])
    else:
        child1_dna_vector += copy(other.dna_vector[last_point:point])
        child2_dna_vector += copy(self.dna_vector[last_point:point])
    last_point = point
```

Two phenotypes can often have different development characteristics. This is the reason why I have chosen to have subclasses of Phenotype for each problem. A typical development method here is just to pass the genome component vector along to the phenotype without any change. If a problem requires something else, you just do it here. A subclass implementation is also usefull when calculating fitness scores. When encountering a new problem you just write a new subclass that inherits some functions and then you just override the fitness function into what the new problem requires. If you need a new selection mechanism, you can incorporate it in the EA algorithm just as the rest of the mechanism. The only requirement is that it return two phenotypes as parents.

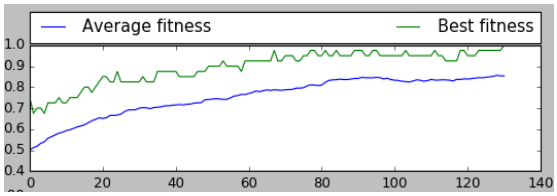
One-Max Problem performance

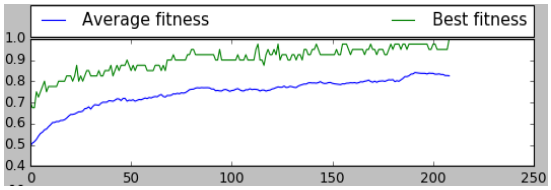
I found that the minimum adult pool size where the problem was solved consistently in under 100 runs was ca 250. This was using a mutation rate of 0.15 and crossover rate of 0.75. The mutation protocol was individual.



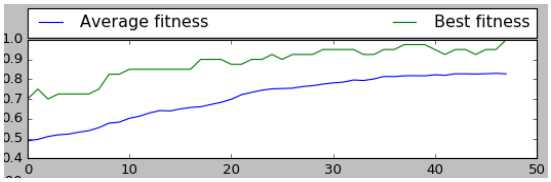


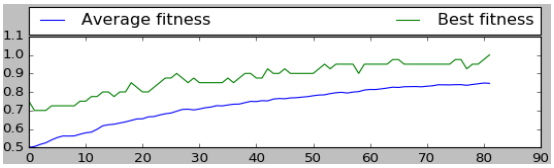
Left: crossover rate: 0.75, mutation rate: 0.15 - avg soultion found after 100
Righ: crossover rate: 0.95, mutation rate: 0.15 - avg soultion found after 80





Left: crossover rate: 0.5, mutation rate: 0.15 - avg soultion found after 109
Right: crossover rate: 0.75, mutation rate: 0.3 - avg solution found after 187

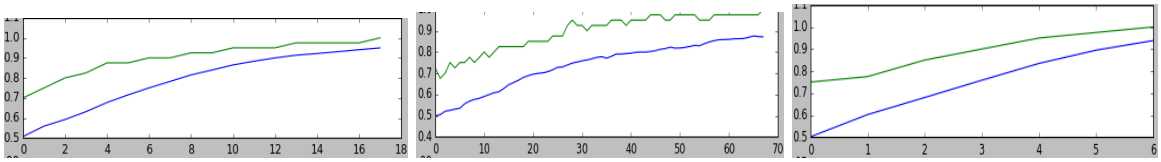




Left: crossover rate: 0.75, mutation rate: 0.05 - avg soultion found after 78
Right: crossover rate: 0.95, mutation rate: 0.05 - avg soultion found after 67

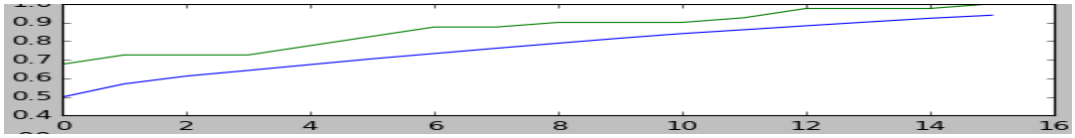
This was the best configuration found. It is important to note that the crossover point was set randomly here. This introduces some randomness, and the mutation becomes less important for exploration.

Parent selection experimentation



Left: Sigma scaling. Avg soultion found after 15 gen.
Center: Boltzmann selection. Avg soultion found after 64 gen.
Right: Tournament selection. Tournament size: 14, “slip-through probability”: 0.25. Avg soultion found after 9 gen. This is easily the best parent selection scheme here.

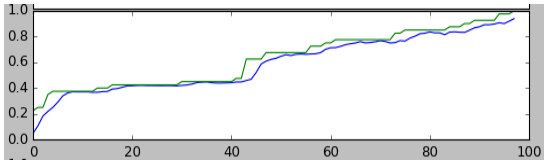
One-Max random bit vector target

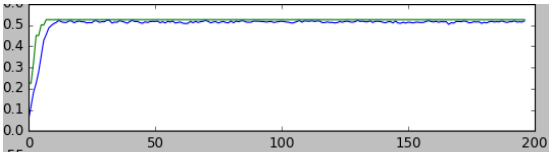


I do not expect this to make the problem harder. A random target is just as hard to find using the same algorithm. The random solution is just as specific as all zeros. I found that the difference was very small, and it can be explained by som randomnes in the implementation.
Fitness proportionatet, crossover rate: 0.95, mutation rate: 0.05 - avg soultion found after 66
Sigma, crossover rate: 0.95, mutation rate: 0.05 - avg soultion found after 15
Boltzmann, crossover rate: 0.95, mutation rate: 0.05. Avg soultion found after 66 gen.
Tournament same as above, crossover rate: 0.95, mutation rate: 0.05-avg soultion found after 9

LOLZ Prefix Problem performance

In 4 out of 8 runs with z=21 the algorithm gets stuck in a local maximum of several leading zeros. It is unable to find the solution. This could be solved by using component mutation instead of individual mutation.

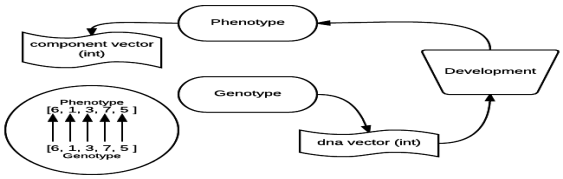




How it looks when a solution is found to the left. Not found to the right.

Surprising Sequence Problem performance

The genetic encoding I have used here is shown in the diagram below. I am using an integer vector in both genotype and phenotype so the development process is not more severe than copying the dna from geno to pheno.



Fitness

I experimented with different methods of calculating fitness. The first implementation was based on checking what the longest substring in a phenotype that was a surprising sequence. This proved very heavy computationaly, so I exchanged it with a simpler count of violations, where one violation is a repetition of a pair with the same distance. The fitness is then calculated as

$$Fitness = 1 - \frac{Nr\ of\ violations}{Max\ nr\ of\ violations}$$

Where max nr of violations in the global problem is given by $n * (n - 1)/2$ and $n - 1$ for the local. Here n is the size of the phenotype.

Local

Symbol size	3	5	10	15	20
Population size	100/200	100/200	390/450	500/600	330/400
Nr of generations	2	55	180	481	3971
size	10	26	100	221	390

Global

Symbol size	3	5	10	15	20
Population size	500/600	200/300	500	500/600	500/600
Nr of generations	1	2	7	20	106
size	9	12	25	37	51

Longest sequence found local:
Symbol size,
3: [1,0,0,2,1,2,2,0,1,1]
5: [2,0,0,2,1,2,3,1,3,4,4,0,1,4,2,4,1,1,0,4,3,0,3,3,2,2]
10:_[5,9,0,5,1,0,8,8,1,8,3,1,6,9,8,4,8,6,8,0,9,9,6,3,5,0,2,6,7,2,2,0,3,8,7,3,4,7,8,9,5,6,4,5,5,2,1,7,6,2,3,9,2,7,4,6,1,1,5,4,2,5,3,0,6,6,5,7,0,0,4,4,9,7,7,1,3,3,6,0,1,9,4,1,2,8,5,8,2,4,0,7,9,3,2,9,1,4,3,7]
15:_[2,6,0,5,13,11,2,3,9,8,0,4,4,1,1,4,7,5,11,11,1,10,1,11,4,3,5,10,9,6,10,6,3,14,3,7,4,10,11,9,10,13,3,2,13,10,14,11,14,4,9,3,1,7,3,6,7,7,2,0,12,4,8,3,3,0,9,0,2,2,7,8,7,9,12,3,13,8,11,7,11,5,5,14,13,12,7,6,14,2,11,0,6,9,13,2,1,2,4,13,9,1,0,0,8,14,14,5,1,9,9,2,8,1,5,7,13,14,12,11,8,6,5,8,10,2,14,1,3,4,6,6,1,14,10,7,0,3,10,10,3,8,5,6,12,1,12,8,9,11,3,11,6,13,6,2,10,8,8,13,5,4,11,13,7,10,4,5,3,12,12,2,9,5,12,6,11,12,14,9,7,14,8,4,12,0,13,13,0,11,10,12,10,0,7,12,5,9,4,0,10,5,2,5,0,14,7,1,8,12,9,14,6,4,2,12,13,4,14,0,1,13,1]
20:_[19,15,4,8,4,15,7,4,2,2,12,15,19,18,17,8,1,1,13,10,8,12,1,12,2,5,7,14,7,13,0,3,5,8,10,1,7,11,6,14,10,18,0,18,12,14,11,14,9,15,5,11,10,5,18,5,3,0,15,2,14,8,8,18,8,17,16,6,8,9,16,19,19,10,2,7,5,15,15,10,10,0,2,19,12,0,8,13,3,10,7,2,11,11,17,3,2,16,13,18,2,17,19,14,18,13,4,3,16,1,0,7,9,10,16,14,17,10,17,11,5,0,19,2,0,14,6,18,6,2,3,6,7,0,11,3,11,13,16,18,7,3,13,19,16,7,18,18,10,9,1,8,7,19,4,1,3,4,17,14,12,16,8,5,6,16,4,12,7,1,19,0,0,9,11,7,10,12,18,4,10,4,7,15,3,18,9,7,6,19,1,15,16,15,6,13,1,4,4,16,11,15,14,5,14,13,17,18,14,0,1,11,1,17,17,4,13,14,15,12,13,5,19,9,4,14,14,2,18,3,14,16,3,9,6,5,1,5,5,12,17,12,5,9,19,7,12,9,2,6,15,9,17,13,13,6,10,13,2,8,16,17,5,10,6,6,11,8,3,17,2,1,14,4,6,4,5,13,12,8,14,3,7,8,6,17,1,6,3,8,15,0,5,17,15,11,9,13,11,4,19,3,1,2,4,0,12,12,10,19,8,2,9,9,14,19,11,18,1,10,11,16,0,4,9,5,2,15,18,16,5,16,9,18,15,1,18,11,19,5,4,11,0,6,12,3,3,19,6,9,8,19,17,7,16,2,13,15,8,11,12,19,13,8,0,10,3,12,6,1,16,10,15,17,9,3,15,13,7,17,6,0,17,0,13,9,0,16,12,4]
Longest sequence found global:
3: [0,2,0,0,1,2,1,1,0]
5: [1,4,3,2,2,4,0,1,0,2,1,3]
10: [4,7,6,3,1,4,3,0,2,0,9,6,8,8,7,5,1,9,2,5,7,3,4,0,1]
15: [10,3,6,4,8,11,14,12,2,9,5,4,7,1,13,13,12,0,13,14,5,9,3,12,6,14,4,3,2,7,10,1,10,8,5,6,11]
20:_[4,13,11,17,18,2,6,7,1,0,14,10,8,10,3,19,0,8,16,2,9,6,3,12,12,4,5,15,16,9,14,18,11,15,8,18,17,1,4,19,1,9,12,7,0,2,13,17,11,5,10]

Difficulty

I think the LOLZ prefix problem is the hardest. This is of course dependant on the parameters of each problem, but generally what we see in this problem is a very high local maximum, that is very hard to get away from. I could have programmed a different fitness function to avoid this, but this would defeat the purpose of the problem. The simplest problem to solve have to be the One-Max problem. This solution landscape is almost without local maximas. The same aplies between global and local surprising sequence. There is a lot more high local maximas in the global problem as it has more constraints. Here you can often get stuck in a position where you only have one violation, but no single value that can be changed to solve the problem. I rank the problem from hard to easy as follows.LOLZ Prefix, Global SS, Local SS, One-Max.