Evolutionary algorithms
IT3708

Nordmoen, Jørgen H.

February 11, 2013

1 Description

My code in this project is designed around a couple of core classes which make out the main backbone. These classes are meant to be extended for further use and as such can not be used by them self. The main work is done in the classes *select_mech.py* and *select_protocol.py*. These two classes deal with selection mechanisms and selection protocols respectively. The main evolutionary loop is done in *evolution.py*, but this code is quite standard and very short. The class *fitness.py* deals with fitness and contains both the main class to be extended and some classes which I needed to solve the *One-Max problem*. Both *genome.py* and *phenotypes.py* deals with the low level representations of genome and phenotypes. For simplicity I have decided that a phenotype class will contain the genome which created it. The next class of some importance is *population.py* which contains a number of phenotypes and some methods for getting statistics from the population. The last file of any major importance is the *main.py* file which contains the starter motor for running all of the classes above. Since I have chosen to have a command-line interface for my code the main file must be changed in order to add new classes. This is because Python is not really that happy about importing files at runtime, and the code is somewhat static in that regard. New configurations must also be added in the main file which is somewhat unfortunate, but the upside is that the code can be very easily run from the command-line with different configurations which make it somewhat easy to automate.

1.1 Selection Mechanisms

Since selection mechanisms are a core part of this project I will here describe most of the core code which would needed to be extended in order to add new code.

Listing 1: Main class which selection mechanisms should inherit from

```
            'not equal the wanted amount, was {0} expected {1}'
                .format(len(selected), amount))
    return selected

class SelectionMechanism(object):
    def __init__(self, eliteism_count):
        self.elite = eliteism_count

    def sample(self, amount, population):
        assert amount > 0, 'The amount to select is to little'
        assert len(population) > 1, 'The population must have
            at least two individuals'
        assert type(amount) is IntType, 'The amount must be an
            integer'
        return self.sub_sample(amount, population)
```

If the new selection mechanism does not need any variables it self, it can contain as little as a new definition of *sub_sample*. This method receives an amount variable and an array of phenotypes. The amount variable signifies

how many individuals this selection mechanism should retrieve. The population is just an array of individuals which this sub_sample method should select from.

This class also contains a few handy functions, but they are of little importance to this report.

1.2 Selection Protocol

The selection protocol is the second important class in this project, it is tasked with creating a new population, by using a selection mechanism to select parents and children.

Listing 2: Main class which selection protocols should inherit from

```
class SelectionProtocol(object):
    def __init__(self, select_alg, num_parents=0.5,
        num_children=0.5):
        assert select_alg, 'The selection algorithm must be
            something'
        assert 0.0 < num_parents, 'Must select some amount of
            parents'
        assert 0.0 < num_children, 'Must select some amount of
            children'
        self.select_alg = select_alg
        self.num = num_parents
        self.num_children = num_children

    def select(self, population):
        assert population, 'The population must contain
            something'
        size = len(population)
        next_pop = self.sub_select(population)
        assert len(next_pop) == size, 'The new population size
            is different than requested'
        return next_pop

    def sub_select(self, population):
        pass
```

In listing 2 we can see that this shares much of the same architecture design as in the previous section. This class expects a selection mechanism class, a float representing the percentage of parents to select for mating and the percentage of children to be created during mating. The last two variables only influences variation and so on, as all selection protocols implemented so far returns the same size population as it is given as input[1].

Listing 3: An example implementation of a selection protocol

```
class FullReplacement(SelectionProtocol):
    def sub_select(self, population):
        amount = len(population) * self.num
```

[1]This is done so that population size is the same all through out a run

```
        selected_mates = self.select_alg.sample(int(ceil(amount
            )), population.get())
        mates = [sample(selected_mates, 2) for i in range(len(
            population)/2 + 1)]
        next_gen = []
        for mom, dad in mates:
            c1, c2 = mom.get_gene().crossover(dad.get_gene())
            next_gen.append(c1)
            next_gen.append(c2)
        while len(next_gen) > len(population):
            next_gen.pop()
        map(lambda x: x.mutate(), next_gen)
        return Population(map(lambda x: x.convert(), next_gen))
```

In the code snippet 3 we can see how a selection protocol is implemented in this project. First a set of parents are selected using the selection mechanism chosen as input. Then we create random couples for mating[2].

1.3 Architecture

The rest of the code follows very much in the same path as the code shown above. By using inheritance we can get a lot of different implementation, which can still be used by code not specially design for the new implementations. As can be seen in the code above I have included a some amount of assertions to try and control what the subclasses do[3]. Most of the files in this project contains one class which is designed to be extended, because of the length requirement on this report I have not included all those classes.

The only code not following this "standard" is the main class which only job is to parse the command-line input and start the evolutionary loop. Because selection mechanisms, protocols and all the variables associated with them can be configured on the command-line this is sort of the weakest link in the project. Since new implementations of selection mechanisms must be included here the main file must also be changed. But this is not something which affects the rest of the code, if I had chosen another user interface style this could be avoided.

2 Code Modularity

The core code in this project is very modular as I hope was somewhat clear in the previous section(1). To expand on that further I will be giving some examples here, but also try and explain the main drawbacks making this somewhat less modular.

Since most of the code is designed around classes most of the job extending this is already done. For future implementations all that is needed is to create

[2]Some other project that I have talked to have used the selection mechanism again when creating couples, I have chosen to keep it this way to maintain some variation. In some of the selection mechanisms this can be controlled, but I think this should not hold this project back that much. If needed this can easily be changed.

[3]By using some optimization flags for Python these will be ignored. This can be handy when wanting to run the code faster

a subclass, implement the function appropriate for that superclass and that is mostly it.

The main thing holding back, as I mentioned in section 1.3, is that the interface needs to be updated to accommodate new classes. This is mainly because Python does not support importing classes at runtime in a more nice manner than what is supported currently. This means that to include new classes one must import them in the main file and then extend the interface to support this. This is quite cumbersome, but the benefits of a command-line interface is better than the alternatives, at least for me.

In the listing 4 we can see how the modularity comes in to play. Since all the other code in the projects expects that a fitness function can just be called, all this new code has to do is to extend the main fitness class. Since this new fitness class is dependant upon a target value it need to define its own constructor, but then all that is needed is to implement "sub_eval".

Listing 4: An extension of the fitness class which support checking against a random target

```
class RandomBitSequenceFitness(BitSequenceFitness):
    def __init__(self, target):
        assert target, 'The target bit sequence is None'
        assert isinstance(target, bitarray), 'The target needs
            to be a bitarray'
        self.__target = target

    def sub_eval(self, pheno):
        assert len(pheno) == len(self.__target), ('The target
            sequence has a ' +
                'different length than the given gene')
        count = 0
        bitArr = pheno.get_value()
        for i in range(len(bitArr)):
            if bitArr[i] != self.__target[i]:
                count += 1
        return count
```

The same is true for all of the other "internal" classes used, to crate a new phenotype all that is needed is to implement the base phenotype and implement or reuse the methods defined within it. The same is true for new genome implementations, the challenge here is new genetic operators. If there is a need for anything other than a reimplementation of crossover or mutation this might be a problem. If the methods can not be called either when crossover or mutation is called, there will need to be some changes to the other code, which of course is unfortunate.

3 Performance Analysis

To test the performance of my code I designed two scripts which automated my analysis. In the file *test_population.sh* there is code for running 20 runs of my code with population sizes from 10 to 20. The code uses different random variables so we can get a proper average. The other settings was "crossover=0.5",

"mutation=0.025", "elite=5" which means that it has a one point crossover, approximately one mutation per gene and the 5 best[4] are used for mating no matter what. I then use another script *average.py* to take the average over all which is then plotted using *plot_population_average.gnuplot*. In figure 1 you can see the result of all the runs[5].

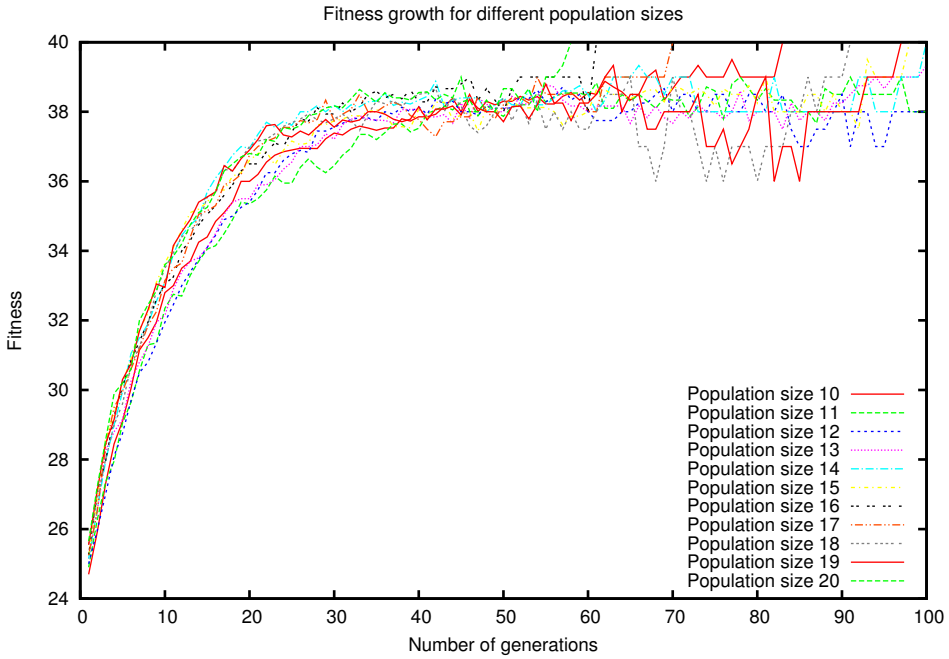

Figure 1: Average fitness growth for the best individual for different populations

As can be seen from the plot there is no clear winner, several reaches the max fitness within 100 generations and some don't. This is also dependant on the random seed selected. For this reason I will continue forward with a population size of 15, because I'm quite sure that it should always make it. On previous runs I have also seen it do quite a bit better so I'm quite confident that 15 is a good size to chose.

When testing for crossover and mutation rate the task was a bit more difficult to plot concise. For this reason I made a another script, *test_crossover.sh*, which can test a lot of different mutation and crossover rates. The problem is plotting this. I haven't found a pretty way of showing it, so below I include two out of seven plots[6], one containing the best results and one showing that it did not improve much. For the plots I have run 20 times with different seeds for each of the 20 times, but the same seed for different crossovers and mutation rates. I used a population size of 15 as said above and elitism of 5 to keep

[4]If this is possible, for some population sizes this means that only the best are used for mating

[5]In many of the plots there is no standard deviation or average plotted, this is done because including them would mean so much clutter that it would be impossible to interpret anything useful from the plots

[6]There should be no problem recreating the results and looking at them

it consistent with the test ran above.

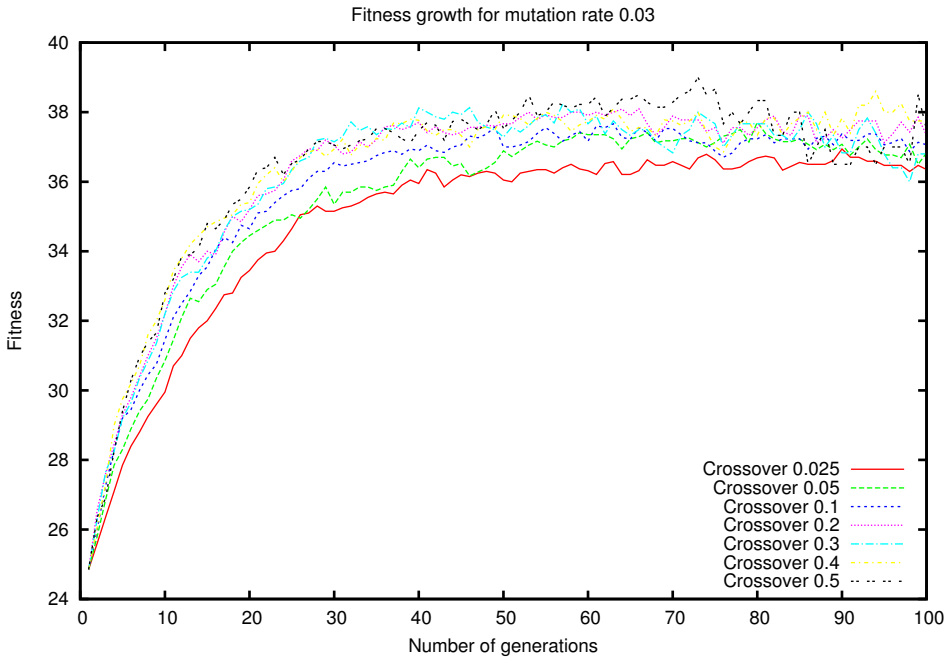

Figure 2: Average fitness for the best individual with a mutation rate of 0.03 and all the different crossover rates tested for

In figure 3 we can see that with a crossover rate of 0.5 and a mutation rate of 0.015 we can solve the *One-Max problem* in around 50 generations[7]

3.1 Selection Mechanism

When testing for the best selection mechanism I again started by making a new script for automated testing, *test_selection.sh*. This files run through all of the different selection mechanisms that I have implemented, namely *Fitness Proportionate*, *Sigma Scaling*, *Rank Selection* and *Tournament Selection*. The result can be seen below in figure 4.

From this we can see that both Sigma scaling and Rank selection do a very good job and improve quite a lot on Fitness Proportionate. Tournament selection does lag a bit behind, but this also have some parameters that would need tuning for better results. For this task I decided to try and tweak them some, but there is room for improvement.

3.2 Random Target

For this task I ran a new script which generated 40bit long random strings and ran four different tests, again with 20 different runs. In figure 5 we can see

[7]This was the lowest of all the seven runs, but random seed will affect this, so this might not be exactly the same for other runs, but I think that average of 20 runs should yield good results

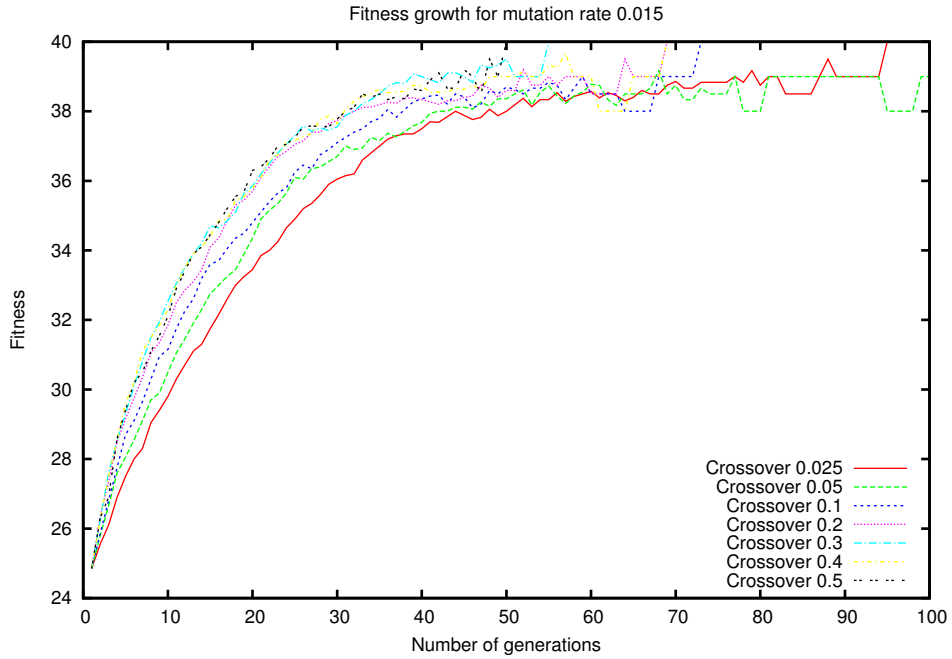

Figure 3: Average fitness of the best individual with a mutation rate of 0.015 and all the different crossover rates tested for

the results. If we compare this with the results obtained in figure 4 we can see that there really is not that much difference between them. Even tough it is a bit quicker in figure 4 that can easily be attributed to random and as such I conclude that trying to evolve a random target is no different from an all one target. This makes sense since in many ways an all one target is no different from a randomly chosen different bit string.

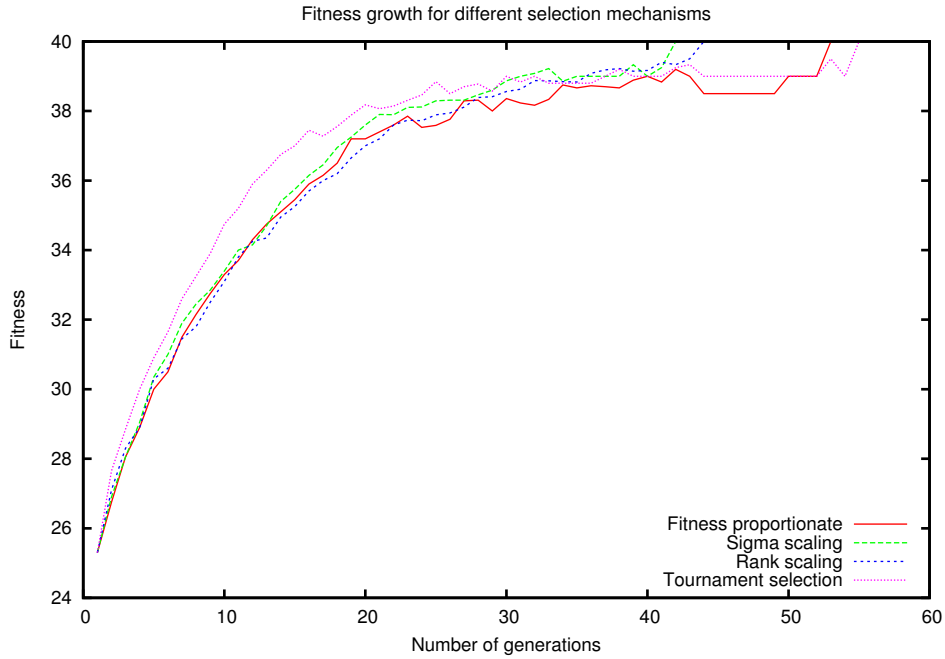

Figure 4: Average fitness of the best individuals with different selection mechanisms

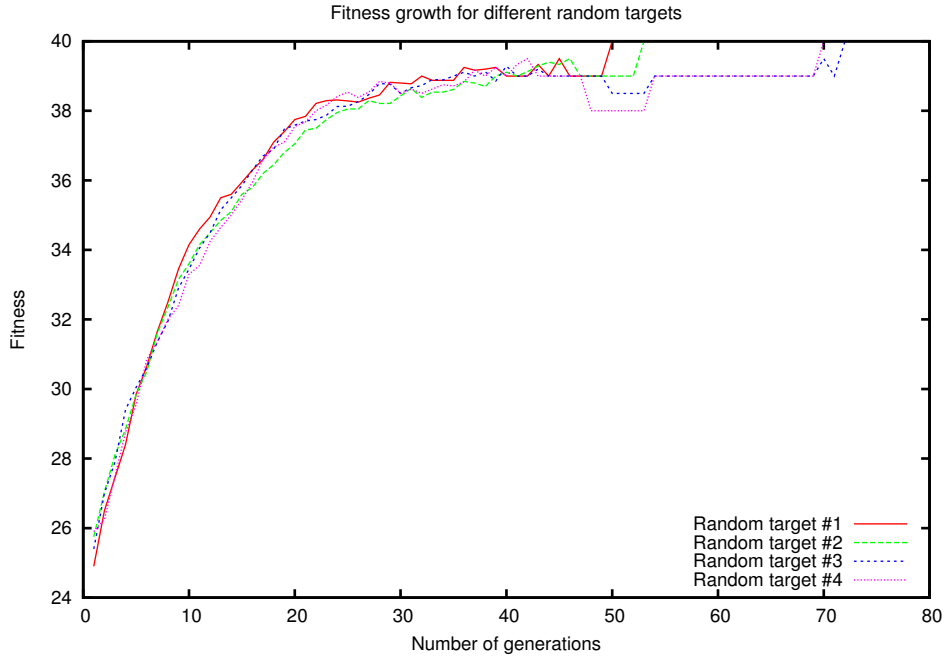

Figure 5: Average fitness of the best individual with four different randomly generated target bit strings