Evolutionary algorithms
IT3708

Nordmoen, Jørgen H.

February 13, 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.

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. I have implemented *Fitness-Proportinate*, *Sigma scaling*, *Rank selection* and *Tournament selection*. These classes just follow the simple description in our provided material.

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.

In select_protocol.py 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 children to be created during mating and the degree of elitism. 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].

In the class *FullReplace* we can see how a selection protocol is implemented in this project. First a set of parents are selected using the selection mechanism chosen. These two parents then get to mate to create two new children. After we have created enough children we have to prune the new population since we are creating a even number of children and an odd might be needed. After this we check if elitism is to be used which then adds the best from the previous population into the newly created one.

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[2]. 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 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.

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

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

In the listing 1 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 1: 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, population):
        assert len(pheno.get_gene()) == len(self.__target), ('
            The target sequence has a ' +
                'different length than the given gene')
        count = 0
        bitArr = pheno.get_gene().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.

For a good example of the resuability we can take a look at the file *blotto.py* which is the extra code I had to add for part B of the project. There really isn't that much new code added to solve a completely different problem. Other than that I had to extend the main class, but this is not an effect of the overall modularity of the code and more the lack of modularity in the user interface.

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 10 runs of my code with population sizes from 10 to 100. The code uses different random variables so we can get a proper average. The other settings was ”cross_rate=1.0”, ”cover_rate=0.5”, ”mutation=0.025”, ”elite=3” which means that it has a one point crossover, approximately one mutation per gene and the 3 best[3] are used for the next population no matter what. I then use an-

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

other 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[4].

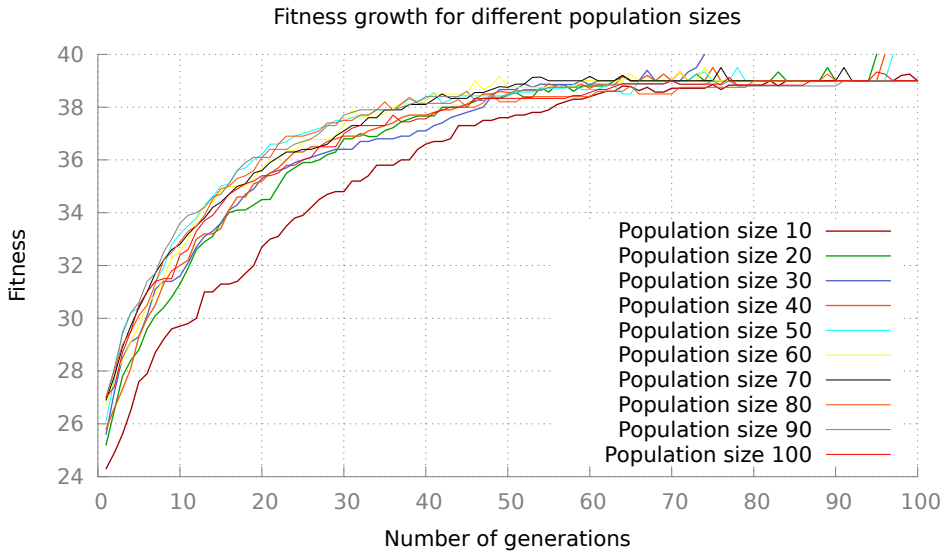

Figure 1: Average fitness growth for the best individual for different populations

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 40, because I'm quite sure that it should always make it. Although this is very hard to say as it seem that with the current configuration it is all rather random if it makes it or not, but as one can see 40 makes it. This means that 40 has done well over 10 runs with a new random seed each time.

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[5], one containing the best results and one other which show another good configuration. For the plots I have run 10 times with different seeds for each of the 10 times, but the same seed for different crossovers and mutation rates. I used a population size of 40 as said above and elitism of 3 to keep it consistent with the test ran above.

In figure 2 we can see that with a crossover rate of 1.0 and a mutation rate of 0.01 we can solve the *One-Max problem* in around 57 generations[6]. To extrapolate from this is quite hard, but it seems that a large crossover rate and a

[4]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

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

[6]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 10 runs should yield good results

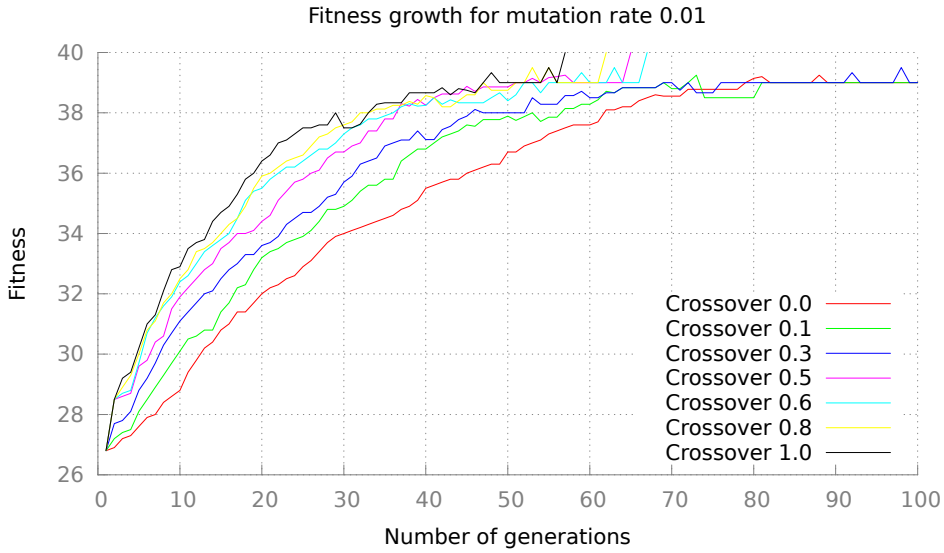

Figure 2: Average fitness for the best individual with a mutation rate of 0.01 and all the different crossover rates tested for

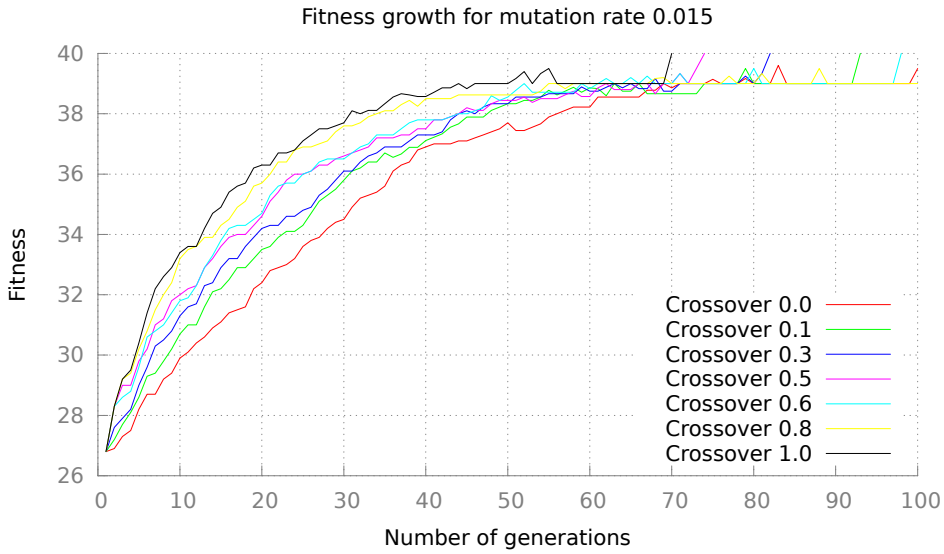

Figure 3: Average fitness of the best individual with a mutation rate of 0.015 and all the different crossover rates tested for

low mutation rate is better than the other way around. This seem to suggest that inheritance is very important, and a low chance of mutation with few major changes to the gene is good. This would correlate well with what we have learned in class about evolutionary algorithms.

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. In ran with the parameters found in the previous section and with tournament size on 10 and the e factor on 0.05 which where both chosen after some testing[7].

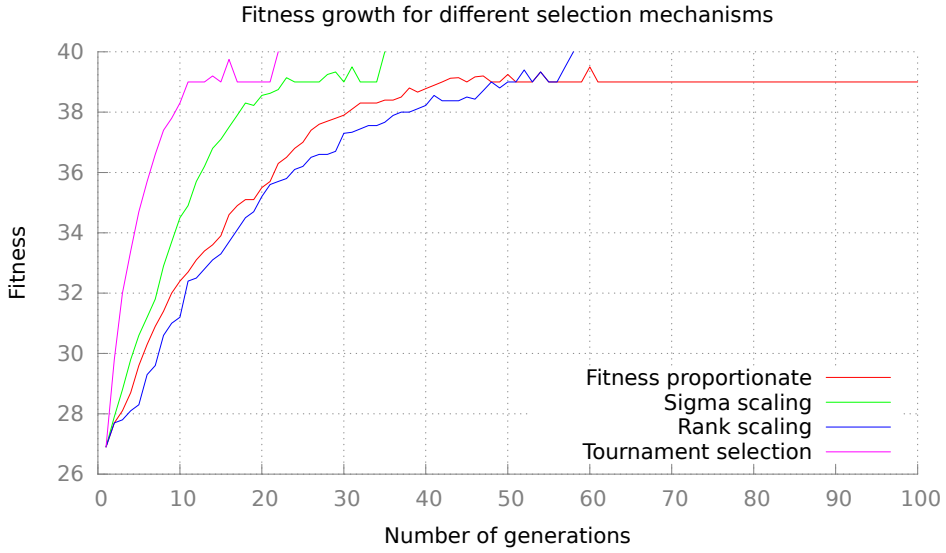

Figure 4: Average fitness of the best individuals with different selection mechanisms

In the figure we can see that Tournament selection does a remarkably good job compared to Fitness Proportionate, Sigma scaling also does a good job and rank selection does improve compared to Fitness Proportionate.

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 10 different runs. In figure 5 we can see the results. If we compare this with the results obtained in figure 6 we can see that there really is not that much difference between them[8]. Even tough it is a bit quicker in figure 5 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

[7] This was not the most extensive testing so Tournament selection might improve more

[8] Both of these runs had the exact same premises which should mean that very little is different between them. One problem is that this is the average over four different random bit strings. Another problem is the inherent randomness which comes with the testing. I ran a couple of times and was quite ready to call One-Max easier, but than I got a couple of results like the ones below and it again switched for me.

from a randomly chosen different bit string. As an example say we have a random target which must have a ”101” somewhere inside and a random phenotype in the population which contains ”111” converting between those two are no different from the one max problem going the other way, and as such is no different.

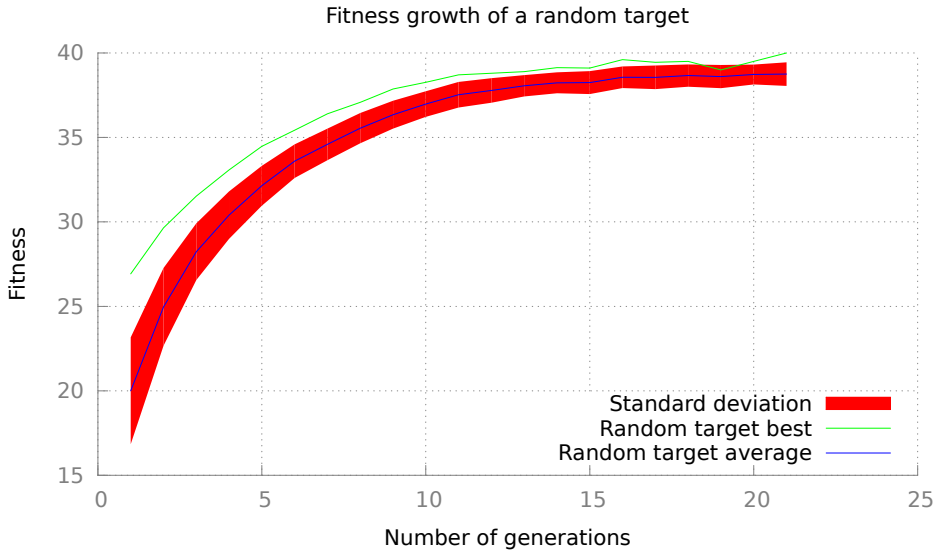

Figure 5: Averaged values of four different randomly selected targets and with average, best and standard deviation.

4 Colonel Blotto

For the representation of Colonel Blotto strategies in a genome I simply used the simple binary genome from part A of the assignment and decided that 5 bits represented one value, to get that into the desired range I divided the value by three. This gives a range of 0 to 10, but it does skew the numbers in favour of 10 slightly. In figure 7 you can see how the genome is sliced to make one value.

To convert the binary genome, which consist of five times the number of battles, into a phenotype containing a list of numbers I just retrieve the bits converting five bits at a time into a number and then dividing by three. Then I sum over all those values and divide each value by the sum thus normalizing them. I keep both the original and the normalized list of numbers in the colonel blotto phenotype.

4.1 Population Entropy

The population entropy is interesting in this problem because it represent the coevolution of the strategies. Since each strategy competes with all the other strategies for dominance the best strategies will most likely mate. It will then

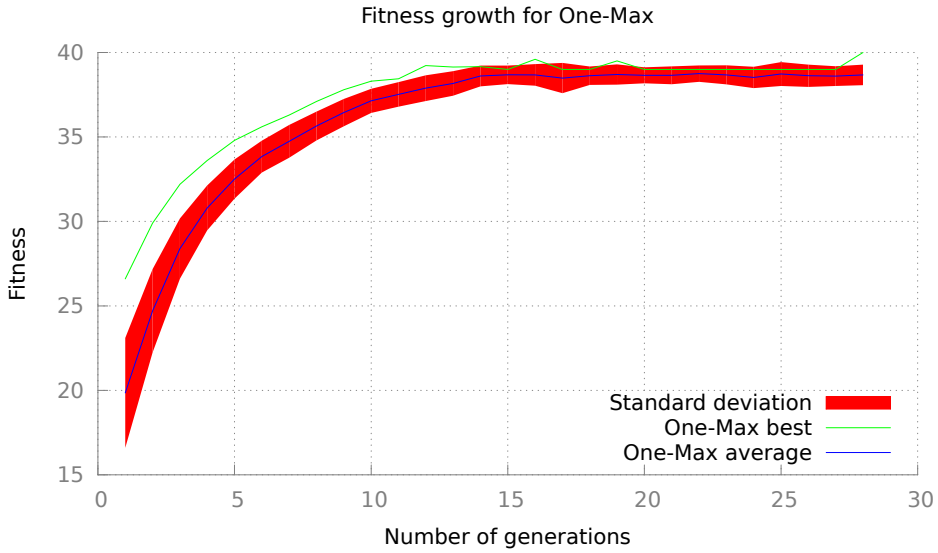

Figure 6: Averaged values for the One-Max problem with average, best and standard deviation.

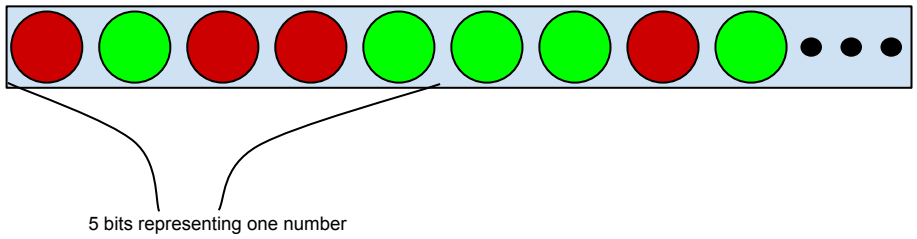

Figure 7: Colonel Blotto genome representation

be better if new strategies takes the best from the previous population, meaning the whole population becomes more homogeneous. In other words each strategy ”tries” to approach the best strategy which means that the average entropy will decrease. This shows off the coevolution taking place through the evolutionary run.

4.2 Blotto Settings

For the runs documented later I tried a few configurations. One problem was that random seed affects the result more then anything so the settings are just chosen on the basis of what I think might work and have seen work in a couple of runs locally.

Variable	Value	Description
Generations	100	I chose 100 since this should be more than enough for the phenotypes to converge and since the assignment states that 100 should be more than enough
Population size	30	I tried with fewer populations and saw a much more rapid growth with 30, even though the assignment states 20 I decided to use 30 since it had work well before and seemed to be good for this problem to
Protocol	full_generational	I used this as it seems to work quite well, the way elitism is implemented also affects this and mens that it does not stray that much away from generational mixing
Mechanism	tournament	Since tournament did so much better than all the other selection mechanisms I really only tested with Tournament selection
–cross_rate	0.8	Even though 1.0 worked well for the One-Max problem for this 0.8 seemed better during testing. This can be explained by the fact that it in some sense increase the elitism, but with a random chance
–cover_rate	0.5	For the cover rate I chose to run with 0.5 because this seem to work well and smaller cover rates have not done particularly well
–mutation	0.01	As mentioned earlier a small mutation rate seem to work well and this also does make some intuitive sense because it means that there won't be to many changes which can effect the fitness of an individual
–elite	3	The elitism was again placed at 3 to get a better result than with no elitism, since this is affected by the chosen protocol I think this makes sense to include some elitism. The number was chosen because it seemed that this was the smallest value which had an effect
–k	10	The size of tournaments was chosen on the basis of how large the total population size
–e	0.05	This value was chosen on the basis that we want the best individuals further, but we also want some variation my thinking here was that this is large enough to include some variation, but small enough to mostly favour the best
–seed	26150	Just a randomly generated seed from $RANDOM

4.3 Blotto Results

In table 1 we can see the results of all the blotto runs. Since all of the runs got the same starting point, only the battle parameters differed between runs, the starting entropy was the same($\tilde{3}.95$). The decision to have the start with the same population always was to ensure that it is only the war parameters that differ in order for us to properly see their results. Note also that the final entropy might not reflect the whole story as the population can diversify later in the run.

Battles	Redeployment	Loss fraction	Description	Entropy(End)
20	0.0	0.0	This run converged quite nicely to one stable strategy, which it did after only about 34 generations.	0.61
20	0.0	0.5	Some what of a periodic shifting.	1.06
20	0.0	1.0	Quite random distribution, but there are some periodicity in that it shifts between distributing for mostly the start or more evenly split among more of the intermittent battles.	2.52
20	0.5	0.0	Quite random, some periodicity late which is lost, but all in all quite random	2.64
20	0.5	0.5	Again quite random, some phenotypes survive a couple of rounds, but the loses to another	1.81
20	0.5	1.0	Some periodicity in that the third is goes from 0.0 to something then back to 0.0 in the last few runs.	1.40
20	1.0	0.0	Mostly random, some survive more than one round and some are quite equal, but no major trends.	2.65
20	1.0	0.5	Mostly random again, few surviving and very few similarities.	3.16
20	1.0	1.0	Pl	0.0
5	0.0	0.0	Pl	0.0
5	0.0	0.5	Pl	0.0
5	0.0	1.0	Pl	0.0
5	0.5	0.0	Pl	0.0
5	0.5	0.5	Pl	0.0
5	0.5	1.0	Pl	0.0
5	1.0	0.0	Pl	0.0
5	1.0	0.5	Pl	0.0
5	1.0	1.0	Pl	0.0

Table 1: Table showing the results