

Tackling the low-autocorrelation problem using a Evolutionary Algorithm

Erik Gast
Klikspaanweg 72 Leiden
+31649994817
erikgast@gmail.com

ABSTRACT

In this paper, I try to solve the low autocorrelation problem using an evolutionary algorithm. I'll do some experiments to find a good configuration for the mutation rate. Also SAG is introduced, which is a replacement for the constant mutation rate. The most important observation from the experiments is that a specialization strategy works the best when searching in large search spaces. When searching in a small search space, a more general search strategy is the better. A few interesting properties of the low autocorrelation problem, the constant mutation rate and the SAG mutation rate will also be discussed.

General Terms

Algorithms, Experimentation.

Keywords

Evolutionary algorithms, low autocorrelation problem, selection, mutation, mutation rate.

1.INTRODUCTION

Because of the difficulty to solve the low autocorrelation problem and it's practical use, it's a interesting to solve this problem. So I try to find a good way to solve the autocorrelation problem, by using an evolutionary algorithm. First I will give a shot description about the low autocorrelation problem in [section 2](#) I'll outline the basics of the algorithm, in [section 3](#), in [section 4](#) the experimental result and in [section 5](#) there is a short conclusion.

2.THE LOW-AUTOCORRELATION PROBLEM

The low-autocorrelation problem for binary sequences is defined as finding the binary string with the maximal objective function.

Feasible solutions: Binary sequences $\vec{y} \in \{-1, +1\}^n$

Objective function:

$$f(\vec{y}) = \frac{n^2}{2 \cdot E(\vec{y})}$$

$$E(\vec{y}) = \sum_{k=1}^{n-1} \left(\sum_{i=1}^{n-k} y_i \cdot y_{i+k} \right)^2$$

So in order to solve the problem you have to maximize the objective function $f(\vec{y})$.

There are already some best know values for binary string of various dimensions, these are given in [table 1](#), below.

Table 1. Best Known Values

n	Best Known f
20	7.6923
50	8.1699
100	8.6505
199	7.835
200	7.4738
201	7.5263
202	7.3787
203	7.5613
219	7.2122
220	7.0145
221	7.2207
222	7.0426

3.THE EVOLUTIONARY ALGORITHM

To solve the low auto-correlation problem, a evolutionary approach is chosen. The advantage of an evolutionary algorithm over an other algorithmic approach is that it has a very simple basic structure. The algorithm consists of a few basic operators: selection and mutation (and evaluation). Because the problem can easily be 'fitted' to an evolutionary algorithm, it might be promising way to solve the problem.

To increase the performance of the algorithm, some of it's variable have to be fine tuned, otherwise it will probability perform really bad. This will be the main challenge.

A basic evolutionary algorithm has a very simple structure consisting of only a few operators. The evolutionary algorithm that is used, uses only selection and mutation (and evaluation).

The evolutionary algorithm looks like this:

```
t = 0
pop = Initpopulation()
for i=1 to maxIterations
    pop'(t) = Mutate(pop't)
    Evaluate(pop'(t))
    pop'(t+1) = Select(pop'(t))
    t = t+1
end
```

3.1 Selection Operator

The selection operator used in the algorithm is based on the tournament selection method [1]. The tournament selection operator selects N (where N is the population size) times 2 random individuals from the current population and selects the fittest of the two individuals. The fittest individual will be an individual in the new population.

```
N = PopulationSize(pop)
for i=1 to N
    indiv1 = RandomIndiv(pop)
    indiv2 = RandomIndiv(pop)
    if Fitness(indiv1) > Fitness(indiv2)
        newPop(i) = indiv1
    else
        newPop(i) = indiv2
    end
end
```

The reason of using the tournament selection is because, it is a simple and fast selection mechanism, and it gives a good result.

A roulette selection mechanism could also be used, but when comparing the two methods, not much differences between the two methods were found, so the tournament selection was chosen.

In this algorithm a (μ, λ) -selection with $\mu = \lambda$ is chosen, but also a (μ, λ) -selection or $(\mu \ll \lambda)$ -selection with $\mu \ll \lambda$ could have been used. A reason not to use the (μ, λ) -selection or the $(\mu \ll \lambda)$ -selection with $\mu \ll \lambda$ is because of its bad runtime performance compared to a (μ, λ) -selection with $\mu = \lambda$. (μ, λ) -selection and the $(\mu \ll \lambda)$ -selection with $\mu \ll \lambda$ generate more offspring than the population size of the parent population and this will drastically decrease the runtime performance. So comparing the selection performance and the runtime performance, a (μ, λ) -selection with $\mu = \lambda$ is chosen.

3.2 Mutation Operator

The mutation operator is one of the most important operators of the algorithm. It mutates the individuals by using a default mutation, which means that every bit of the individual has a probability p to mutate (flip the bit). The mutation operator produces the same amount of mutated individuals as there are in the parent population.

The probability p is the most important variable of the evolutionary algorithm. If you use a large or a small p , then

the algorithm will probably perform bad and might not find the global optimum and even not find the local optimum. So it's very important that the probability to mutate is fine tuned, so that the algorithm performs optimal.

When using a large probability, then running the algorithm will be almost the same as randomly generating all the individuals. If a small probability is used, then it will be hard to get out of a local optimum and therefore it will be very hard to reach the global optimum. So fine tuning the probability to mutate, is very important.

To get these 'fine tuned' p 's, some experiments are done with strings of different dimensions. You have to do the experiments for every string dimension, because there can be a 'big' difference between an optimal p when using for example a string dimension of 5 and when using a string dimension of 20.

Some experiments are done to find 'good' p values for lower and for higher dimension strings.

3.3 Population Size

Because the algorithm is an evolutionary algorithm, it focuses on generating new generations, by the use of mutation. In contrary to genetic algorithms [3], which focuses on crossover and having a large initial population. So because it's an evolutionary algorithm it will have a relative small population size and relative many iterations (new populations). In the experiments, a population size of 15 individuals is used. This is mainly because of runtime performance issues (time issues), normally you would also look for the optimum population size.

4. EXPERIMENTS & RESULTS

In order to fine tune and therefore increasing the performance of the algorithm, a few experiments are done. These experiments consist of finding the best (good) mutation rate and dealing with 'getting stuck' in a local optimum. So some experiments with different mutation rates and functions are done.

4.1 Constant mutation

First, it's interesting to see how well a mutation with a constant mutation rate performs. It's obvious to see that the performance of the algorithm depends a great deal on the mutation rate, so we might suspect that there will be a significant difference in performance while varying the mutation rate.

In the tests, the mutation rate varied from 0.3 to 0.001 with a stepsize of $((0.3 - 0.001)/100)$ and a total steps of 100. The average of the fittest individuals are calculated using a session of 10 runs. Also the average iterations of the fittest individual found in the population, the fittest individual of the 10 populations and in how many iterations the best individual has been found, are calculated.

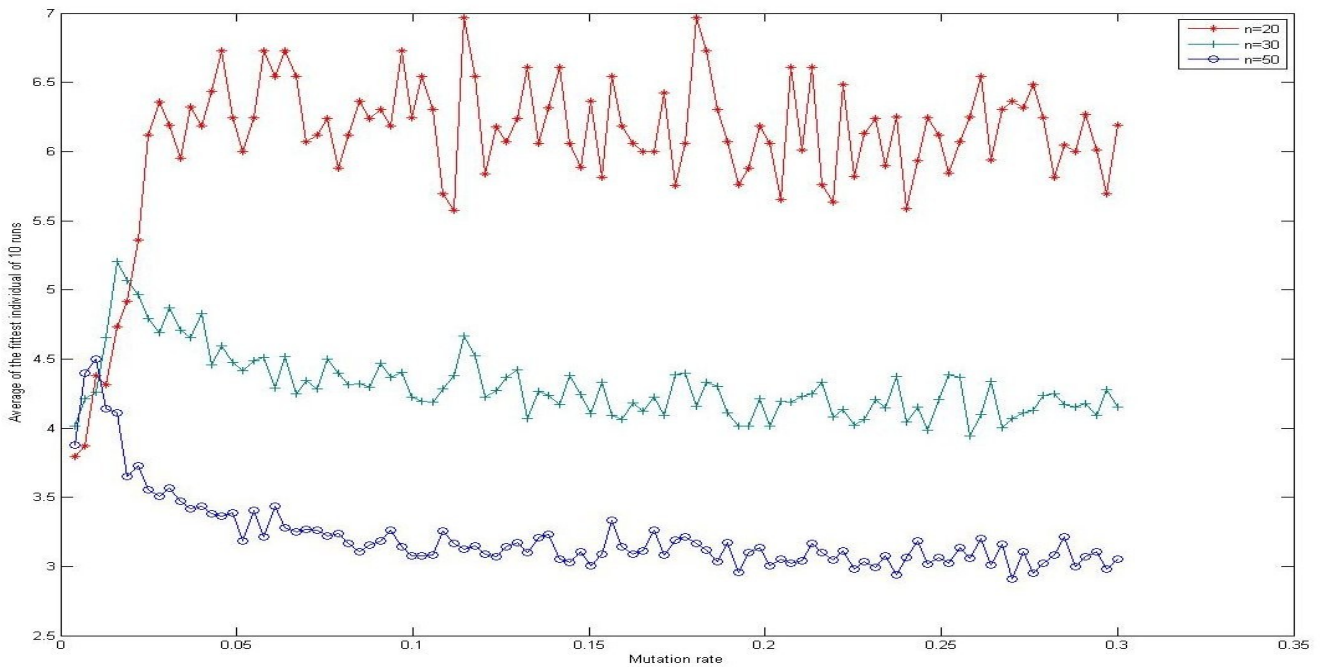


Figure 1. The average performance of the constant mutation rate.

These tests have been performed on strings with a dimension of 20, 30 and 50. You can see the results in [figure 1](#).

As you can see, there is a big difference in variation between the three dimensions. You notice that the average fittest of the one with string dimension 50 ($n=50$) are closer to each other than the average fittest of string dimension 20. This is most likely caused by the fact that there are less local optimums and these optimums are farther away from each other in the one with $n=20$ than the one with $n=30$ or $n=50$. The optimums in higher dimensions are closer to each other. This isn't very surprising, we already knew that it would be harder to find an optimum for string with large dimensions. One other interesting thing is that it converges to a certain objective function value when increasing the mutation rate. For a string dimension of 50 it converges to an average fitness of 3.0, but for a string dimension of 20 it converges to a value of 6.1. This means that when increasing the mutation rate towards a more random mutation (larger mutation rate), the performance will decrease and you will probably not find the local optimum, it will just 'jump' over it. It will only find fitness values which are very common in the search space.

Another interesting part of the graph is the beginning where a very low mutation rate is used (mutation rate < 0.05). You see that for strings with a higher dimension, a small mutation rate is very beneficial. How higher the dimension of the string, the smaller the mutation rate must be (but not too small > 0.004) to perform well. You can explain this by the fact that it's better to find the local optimum (a more specialized search using a small mutation rate) than to do a more random search (large mutation rate) which will most likely not succeed in finding a good optimum, this is because of the large search space it has to search in.

So when searching in large search spaces, it's better to do specialization to find the local optimum, than to do a more global search. And if you use small search spaces it's better to do a more global search than a more specialized search.

It looks like small mutation rates are the way to go, but this isn't really true. When using a small mutation rate you will find a local optimum very quickly (in a few dozen iterations). But we are not

really interested in the local optimum, we want to find a global optimum or at least a local optimum which is close to the global optimum. If you use only a small mutation rate it isn't very likely that you'll ever find the global optimum, in fact it's more probable that you won't find the global optimum at all (when searching in relative large search spaces). You'll need some mechanism to 'jump' out of the local optimum so you have a better chance in finding the global optimum.

4.2 Getting out of a Local Optimum

As discussed, using a small mutation rate does probably only find a local optimum. An interesting experiment will be to see if it's likely/possible to jump from a local maximum to another local optimum. This is interesting because we can use this to improve our algorithm.

As you can see in [figure 2](#), you will probably not jump to another local minimum, when using a small constant mutation rate. If you use a small mutation rate (< 0.08) you will find the local optimum

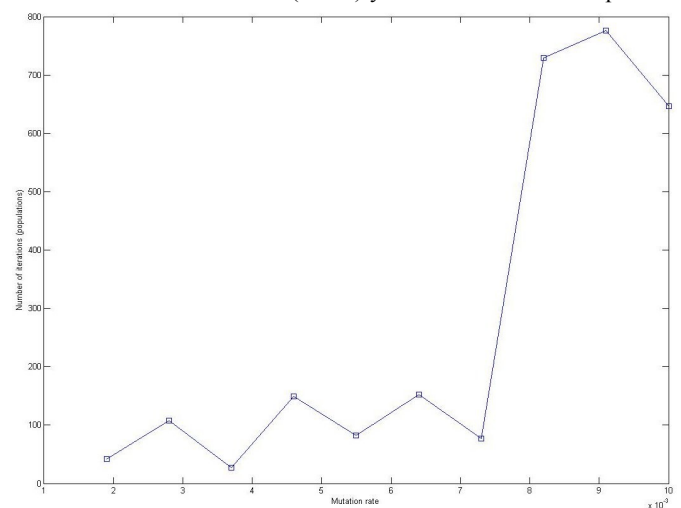


Figure 2. Average number of iterations to find the optimum using 1000 iterations (generations)

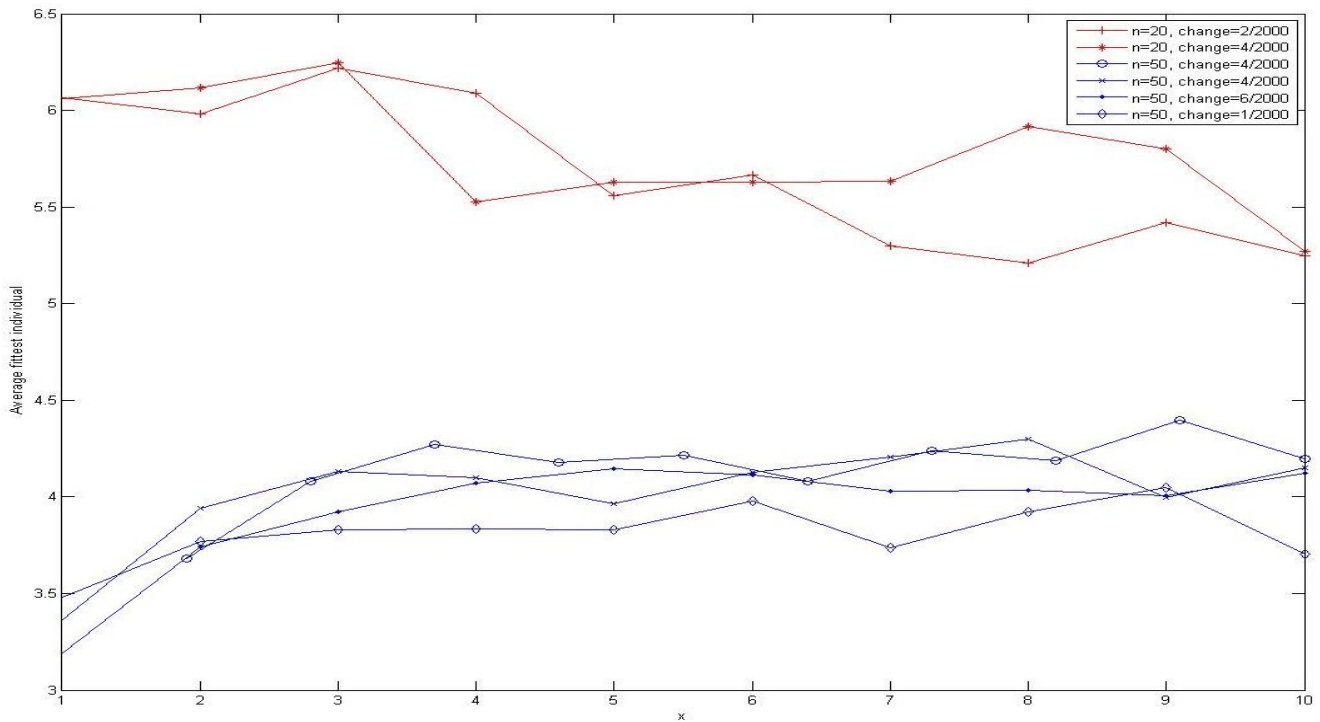


Figure 4. The Average fittest using SAG mutation

very quick, but after you found the optimum, it won't improve in a 'long' time. We can use this observation in our SAG mutation, described in the section below.

4.3 Specialization after Globalization (SAG)

For the second type of experiment an other kind of mutation rate is used. Now the mutation rate isn't constant but will start with a relative large mutation rate and it converses towards a small mutation rate. The idea behind this is that you first need to do some global search and then try to specialize (improve) the global search to find the local optimum. The function that is used is:

$$g(t, x, m, c) = t - (((i/t)^x) * (m - c)) + c$$

Where t is the maximum number of iterations (when it terminates), i is the current iteration, x is number which is responsible for the specialization rate and c is a constant number.

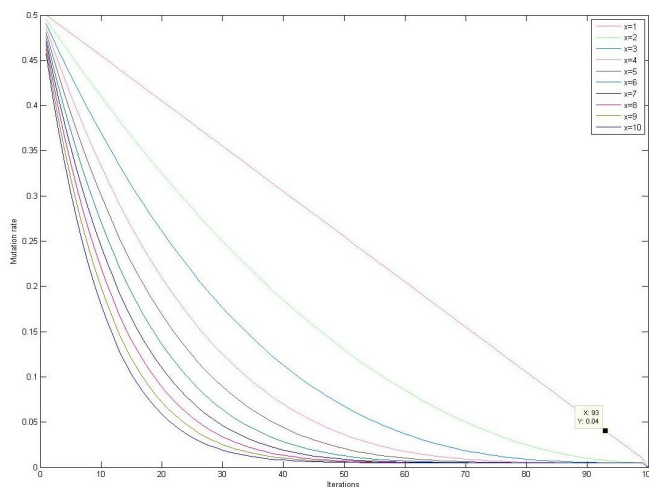


Figure 3. SAG Mutation rate

If you vary x , you will vary how 'fast' it will go into specialization. If $x = 1$, then there will be as much global search as specialized search. If $x = 10$, then there will be a lot of specialization compared to global search. The constant c is the value to which it will converge. In the experiments, a c value of 0.005 is used. In [figure 3](#) you see the mutation rates $g(t,x,m,c)$ produces.

To make this method more effective we can do a sort of mutation rate reset. The mutation rate will be reseted ($i=0$) after a certain interval, so the algorithm will go searching somewhere else to find an other optimum. This method makes use of the observation described in [section 4.2](#), which states that if you're specializing (mutation rate is very small) and the fittest individual does not change, than you probably found a local optimum. So if you found this optimum it is best to 'jump' out of the local optimum and find other (possible better) optimums.

In order to make this method perform well, a good ratio between x , c and the change interval must be found.

In the experiments, x will be varied from 1 till 10 and also the change interval (when the mutation rate is reseted) will be varied and it's tested on strings with dimensions of 20 and 50.

In [figure 4](#) you'll notice the slidely different graph shapes when using strings with dimension 20 and 50. The graphs created by using a dimension of 20 perform better if x is small and perform worse when increasing x . This means that a more random exploration of the search space is a better strategy when searching in a search space with string dimension 20.

If you use strings with a higher dimension (in this case 50) you see sort of the reversed effect. Here the algorithm performs worse when using a small x and better when using a larger x value. So specialization is a better strategy than exploration when using strings with higher dimensions. How much you specialize depends on the ratio between the change interval, the x value and the begin ($m-c$) and end (c) mutation rate.

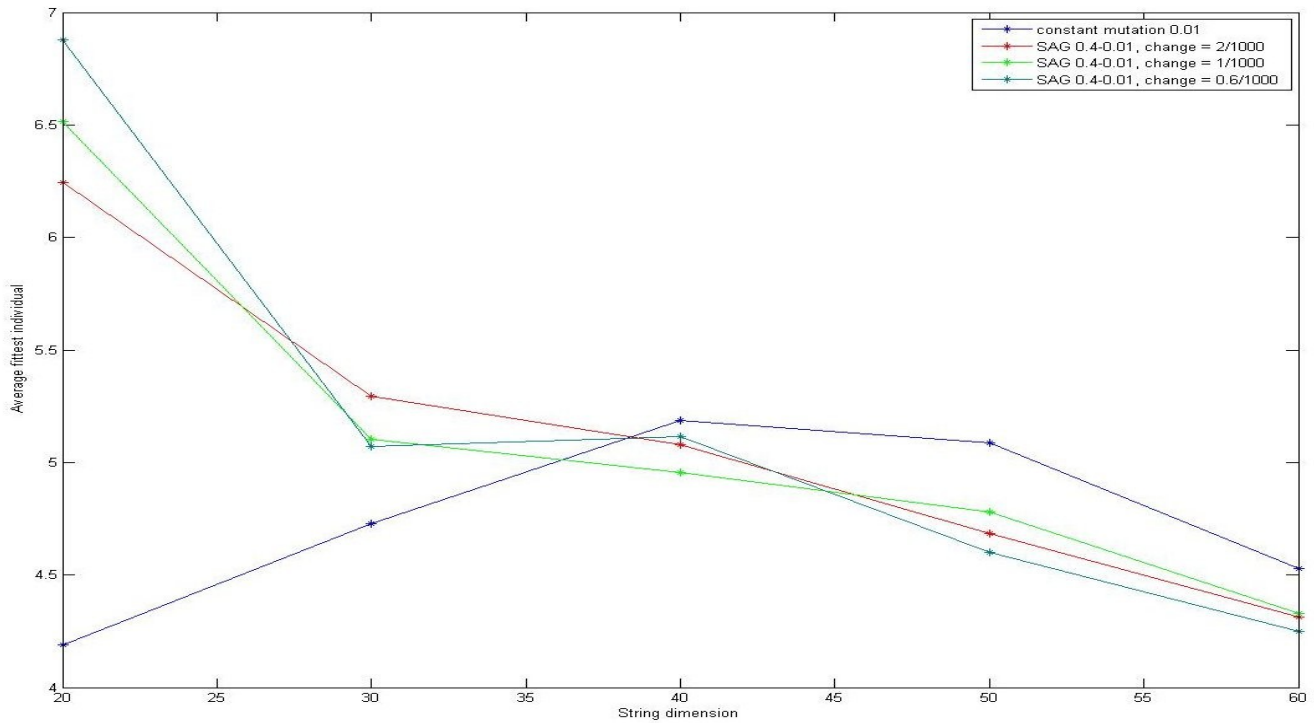


Figure 5. Average fittest in different string dimensions

This are the same results as concluded from the constant mutation rate experiment. Now we can say that in order to search in a large search space, it's better to spend more time on specialization than on a more global search.

4.4SAG vs. constant mutation

To see how good each method performs compared to each other, some experiments are done with the two methods and different string dimensions.

The constant and SAG mutation are tested against strings with dimension 20, 30, 40, 50 and 60. To calculate the average, a session of 20 runs is used. One run consists of 10000 iterations

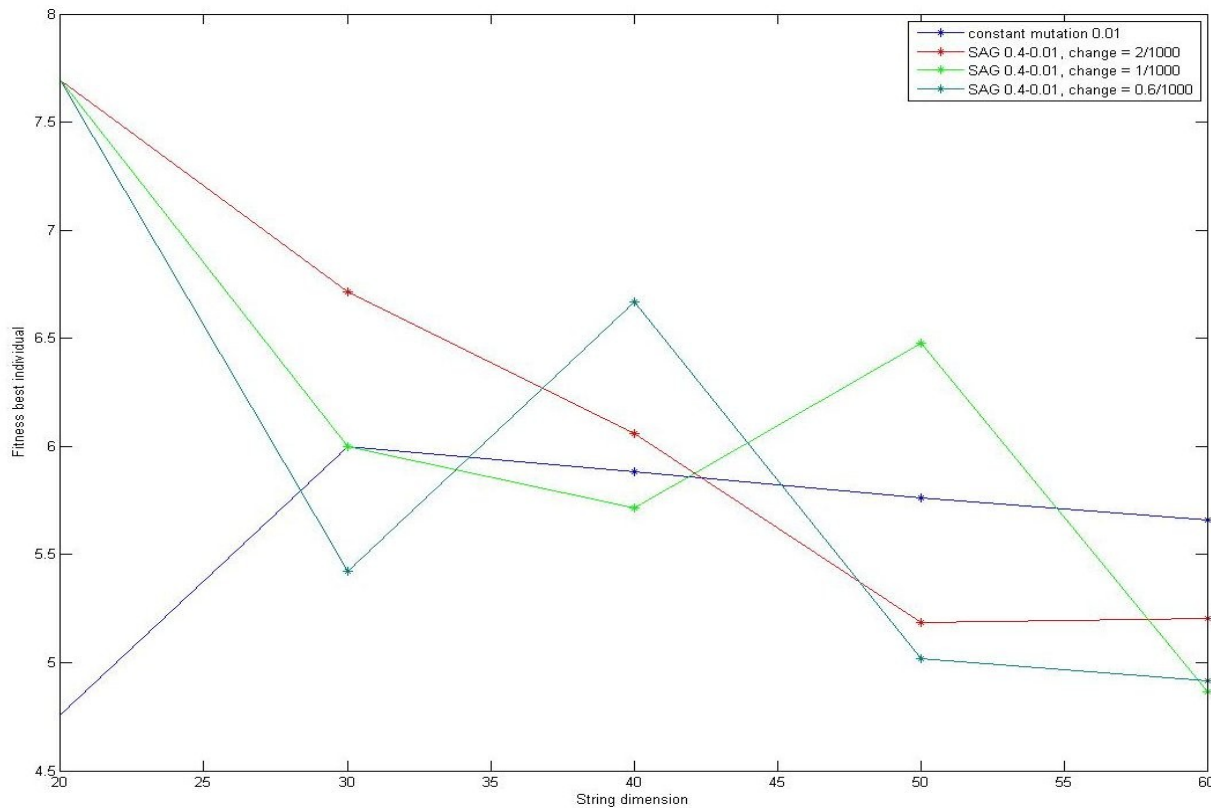


Figure 6. Best individual found in different dimensions.

and a population size of 15. A constant mutation rate of 0.01 and a SAG mutation of 0.4 to 0.01 ($m = 0.399$, $c = 0.01$) with $x = 9$ is used. The results are shown in [figure 5](#) and [6](#).

One thing you immediately notice in [figure 5](#) is the graph of the constant mutation, it performs surprisingly well. The SAG does not perform (on average) that well on strings with dimensions higher than 40 compared to the constant mutation rate. But it does perform better than the constant mutation rate (when the mutation rate is 0.01) on strings with dimensions smaller than 30. This actually contradicts our previous findings. An explanation for this contradiction is, that not enough generations (iterations) have been generated and specially for the SAG method. The SAG method needs a fair amount of generations in order to perform well on higher dimension strings. So my expectation is that the SAG method will outperform the constant mutation rate, if enough generation are generated, based on the previous findings.

Because the average performance of the best individual in population might not be that important. Actually more important is to see which algorithm finds the best optimum. In [figure 6](#) you see which method finds what optimum. As you can see, the constant mutation has a quite steady decrease, while the SAD method fluctuates much more. This is because of the local optimum discussed in [section 4.2](#). The constant mutation is not very likely to jump out of this maximum, whether the SAD method will jump out if this local maximum. And therefor the SAD method finds better optimums in the higher dimension strings than the constant mutation ([figure 6](#)).

The best value found for strings with a dimension of 50 was found with the SAD method. It found a string with the objective function of 6.4767 ([figure 6](#)). This value was found in a 0.2M iterations session. Maybe an even better value could have been found if more iterations were used, but because of the time issue, this was not done.

5.CONCLUSION

The low autocorrelation problem is still a very hard problem to solve. The use of evolutionary algorithms does make it a little easier, but is still unable to solve the problem for higher dimension strings. I have shown the importance of fine tuning the mutation rate and a solution for the 'get stuck in a local optimum' problem.

Using the SAG method does increase the chance of finding a better optimum when searching in large search spaces, but this method is far from perfect. The effect of the variables in the SAD method must be further explored in order to find the best configurations.

If you are searching in large search spaces, the best strategy to use is specialization.

6.FURTHER WORK

Although this algorithm performs relatively well, it is very likely that the algorithm could be improved. To improve the algorithm to performs better and faster, other techniques should be explored. Using an other selection and/or mutation method might increase performance. But the hardest part, fine tuning the parameters will give the biggest performance increase. Tuning these parameters might be done using an other evolutionary algorithm or by using a self-adapting mechanism. More research needs to be done.

7.REFERENCES

- [1] http://en.wikipedia.org/wiki/Tournament_selection
- [2] http://en.wikipedia.org/wiki/Fitness_proportionate_selection
- [3] http://en.wikipedia.org/wiki/Genetic_algorithm
- [4] <http://www.liacs.nl/~baeck/EA/>