Evolving Random Sequences Using a Genetic Algorithm

Abhinav Aggarwal
University of New Mexico, Albuquerque, NM USA

Abstract

This report presents the experimental results with methodology and interpretation as prepared by the author for the extra credit project in the Complex Adaptive Systems (CS523) class at the University of New Mexico in Fall 2015 with Prof. Stephanie Forrest. The study is on using a genetic algorithm to see if a binary random sequence generated using the Python 2.7.10 `randint(0,1)` function can be made more random with respect to a preselected set of tests of randomness and independence. The results show that with a high crossover rate, tournament selection evolves a given population of random sequences into a set that contains all random sequences that pass the tests of randomness and independence with high p-values. The intriguing question that arises here asks why mutation has a negative affect on this evolution.

1 Introduction

Genetic algorithms are already popular for having the ability to evolve patterns and characteristics in a population of individuals which has a diverse set of features. However, can this superpower be exploited to overcome some drawbacks of our state of the art random number generators? In other words, while one knows that all that a genetic algorithm does is a smart random walk over the state space of the genetic encoding of individuals, then can this smartness be used to produce the exact opposite of hunting patters, i.e., discovering a population of individuals that apparently do not have any obvious patterns in it? This report seeks an answer to this question by trying to evolve random binary sequences generated by the Mersenne twisters of Python 2.7.10 into a set of sequences that pass a predefined set of tests of randomness and independence with higher confidence interval than what we started with.

A fundamental question to answer in this experiment is about its credibility and need. Do we really need to do this to generate random sequences when we already have state of the art random number generators? Moreover, is the overhead of running a genetic algorithm worth it? As it turns out, we can infact use genetic algorithms to our advantage so that the overhead of the runtime can be compensated with better quality random sequences. A big advantage here is that even deterministic random number generators (like the Mersenne twistors) whose output depends solely on the seed provided now cease to show this dependence on the seed. For applications where we are concerned with the secrecy involved with this seed, such an approach seems to help. Consider the case where such seed-sensitive RNGs are used to encrypt messages in a highly adversarial environment. In this scenario, compromising the seed has broken down the entire privacy model since the random number sequence can now be completely determined. Using an approach similar to this report, even compromising the seed will not help, since the evolved population does not reveal anything about it.

For the graphs and simulation, I use Python 2.7.10 and the `skidmarks` toolkit, on a Mac desktop, with OS X Yosemite 10.10.5 that has two 2.93 Ghz 6-Core Intel Xeon processors and a 32GB RAM.

This report is organized into various sections. Section 2 enumerates the tests of randomness and independence used in this report to test the quality of random sequences. Section 3 provides details of the genetic algorithm used, with various design decisions made along the way. Finally, section 4 discusses the results obtained and the conclusion for the report.

2 Tests of randomness

Tests of randomness are statistical and emperical measures of how random a given sequence is. Although the notion of randomness is not completely defined or even understood, these tests of randomness try to capture our intuitive understanding of how we think a random sequence should look like. With this view, the tests used in this report form the underlying basis of the fitness function of the genetic algorithm. The better a given binary string performs with respect to a test, the higher fitness value is allocated to it. In this experiment, I use six different tests, out of which some are tests of independence and the others are for randomness. The former is required to make sure the subsequences of the given sequence do not have a high correlation.

2.1 Significance of p-value

In statistics, the p-value is a function of the observed sample results (a statistic) that is used for testing a statistical hypothesis. More specifically, the p-value is defined as the probability of obtaining a result equal to or "more extreme" than what was actually observed, assuming that the hypothesis under consideration is true. Here, "more extreme" is dependent on the way the hypothesis is tested. Before the test is performed, a threshold value is chosen, called the significance level of the test, traditionally 5% or 1% and denoted as α. If the p-value is less than or equal to the chosen significance level (α), the test suggests that the observed data are inconsistent with the null hypothesis, so the null hypothesis must be rejected. However, that does not prove that the tested hypothesis is true. When the p-value is calculated correctly, this test guarantees that the Type I error rate is at most α. Since p-value is used in frequentist inference (and not Bayesian inference), it does not in itself support reasoning about the probabilities of hypotheses but is only as a tool for deciding whether to reject the null hypothesis.

In our experiment, the null hypothesis is that the input sequence contains independent elements with each subsequence independent of the others, with respect to 5% significance level. A drawback of using p-values directly in our fitness function is that they cannot be given a frequency counting interpretation since the probability has to be fixed for the frequency counting interpretation to hold. In other words, if the same test is repeated independently bearing upon the same overall null hypothesis, it will yield different p-values at every repetition. Nevertheless, an instantiation of this random p-value can still be given a frequency counting interpretation with respect to the number of observations taken during a given test, as per the definition, as the percentage of observations more extreme than the one observed under the assumption that the null hypothesis is true. Hence, as a starting point, it seems justified to use p-values as a fitness measure for random sequences in our experiment.

2.2 Chi square test

A chi-squared test is any statistical hypothesis test in which the sampling distribution of the test statistic is a chi-square distribution when the null hypothesis is true. Chi-squared tests are often constructed from a sum of squared errors, or through the sample variance. Test statistics that follow a chi-squared distribution arise from an assumption of independent normally distributed data, which is valid in many cases due to the central limit theorem. A chi-squared test can then be used to reject the hypothesis that the data are independent. The version of chi-squared test I use in this experiment is called the Pearson's chi-squared test, which is applied to sets of categorical data to evaluate how likely it is that any observed difference between the sets arose by chance. It is suitable for unpaired data from large samples. It tests a null hypothesis stating that the frequency distribution of certain events observed in a sample is consistent with a particular theoretical distribution. The events considered must be mutually exclusive and have total probability 1. A common case for this is where the events each cover an outcome of a categorical variable. A simple example is the hypothesis that an ordinary six-sided die is "fair" (i.e., all six outcomes are equally likely to occur). To make sure this test performs nicely on our sequences, we make the length of each random sequence as well as the population size large enough so that much more than the required minimum (5) samples fall inside each category. We choose individual random binary sequences of length $10,000$ each and the total population size of 1000.

2.3 Wald-Wolfowitz test

The runs test (also called Wald–Wolfowitz test after Abraham Wald and Jacob Wolfowitz) is a non-parametric statistical test that checks a randomness hypothesis for a two-valued data sequence. More precisely, it can be used to test the hypothesis that the elements of the sequence are mutually independent. A "run" of a sequence is a maximal non-empty segment of the sequence consisting of adjacent equal elements. For example, the 22-element-long sequence "++++—+++–++++++—-" consists of 6 runs, 3 of which consist of "+" and the others of "-". The run test is based on the null hypothesis that each element in the sequence is independently drawn from the same distribution.

2.4 Autocorrelation test

Autocorrelation, also known as serial correlation or cross-autocorrelation, is the cross-correlation of a signal (in this case random sequences)with itself at different points in time. Informally, it is the similarity between observations as a function of the time lag between them. It is a mathematical tool for finding repeating patterns, such as the presence of a periodic signal obscured by noise, or identifying the missing fundamental frequency in a signal implied by its harmonic frequencies. It is often used in signal processing for analyzing functions or series of values, such as time domain signals. The traditional test for the presence of first-order autocorrelation is the Durbin–Watson statistic or, if the explanatory variables include a lagged dependent variable, Durbin's h statistic. The Durbin-Watson can be linearly mapped however to the Pearson correlation between values and their lags **??**.

2.5 Serial test

This is a 2-dimensional version of the chi-squared test to test independence between successive observations, which can be easily generalized to higher dimensions.

2.6 Gap test

This test is used to examine the length of "gaps" between occurrences of samples in a certain range. It determines the length of consecutive subsequences with samples not in a specific range. For each U_j in a certain range, this test examines the length of the 'gap' between this element and the next element to fall in that range, and hence the name of the test. So, if α and β are two real numbers such that $0 \leq \alpha < \beta \leq 1$, we're looking for the length of consecutive subsequences $U_j, U_{j+1}, \ldots, U_{j+r}, U_{j+(r+1)}$ such that U_j and $U_{j+(r+1)}$ are between α and β but the other elements in the subsequence are not (this is a gap of length r). We would then perform a chi-squared test on the results using the different lengths of the gaps as the categories.

2.7 Entropy

Entropy (more specifically, Shannon entropy) is the expected value (average) of the information contained in each message. 'Messages' can be modeled by any flow of information, which in our case is the random inpt binary sequence. Units of entropy are the shannon, nat, or hartley, depending on the base of the logarithm used to define it, though the shannon is commonly referred to as a bit. Our use of entropy as a test of randomness has to do with its relation to compressibility of random sequences. If a compression scheme is lossless—that is, you can always recover the entire original message by decompressing—then a compressed message has the same quantity of information as the original, but communicated in fewer characters. That is, it has more information, or a higher entropy, per character. This means a compressed message has less redundancy and hence, more randomness.

3 The genetic algorithm

Having described the tests of randomness used as fitness measures for individuals in our population of random binary sequences, this section looks into the details of the genetic algorithm used to evolve the population. Some major design decisions have also been justified along the way. To start with, the popuation of random sequences is obtained from Python 2.7.10's `randint(0,1)` function from the `random` library. Each call to this function is a Mersenne twistor pseudo random number generator call, that returns a uniformly distributed random number between 0 and 1, which is then converted by this function into an integer value. Once a population has been obtained, we compute the fitness of each individual in the population and run tournament selection on the same. The selected individuals are then subject to crossover and mutation and the resulting offsprings form the next generation of individuals. The process is repeated for some number of generations (in our experiments, the number of generations ranges from $100 - 150$) until a population of individuals with the desired properties is obtained.

3.1 Fitness Function

The fitness function in this experiment is defined as the product of p-values each of the six tests of randomness yields for the given binary sequence. This is done to ensure that all tests are passed for the sequence with high p-values. Even if one test yields a poor result, the entire fitness value gets reduced. However, a subtle point needs to be noted here. Even if all tests yield a high p-value, say 0.9 each, the product function yields a fitness of $0.9^6 \approx 0.565$, which means that a fitness of more than 0.56 is considered a very good performance on all the tests. Fig.1 shows the performance of the GA on a population of 1000 random sequences of size $10,000$ with 99% crossover rate and 0.01% mutation rate. As can be seen, the population evolves to contain individuals with fitness values as high as 65% minimum. This is an encouraging result that speaks in favor of the goals of this experiment.

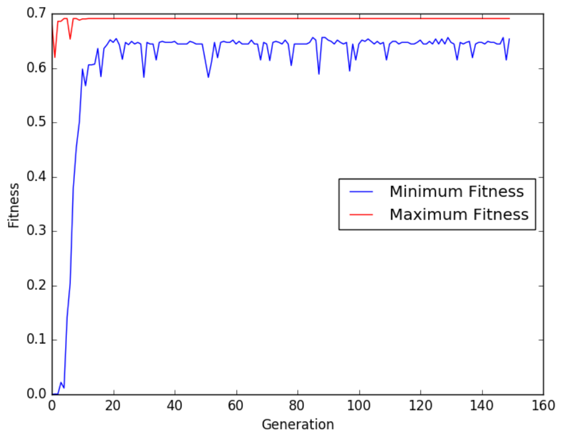

Figure 1: Evolution of random sequence population by the genetic algorithm

3.2 Selection

The selection criterion used here is tournament selection of size 2 or higher. It involves running several "tournaments" among a few individuals (or 'chromosomes') chosen at random from the population. The winner of each tournament (the one with the best fitness) is selected for crossover. Selection pressure is easily adjusted by changing the tournament size. If the tournament size is larger, weak individuals have a smaller chance to be selected. Hence, to not encourage elitism in the experiment, the tournament size is kept fixed at the lowest value of two. However, something interesting happens when there is no tournament selection, i.e. when the individuals are allowed to mate and produce offsprings irrespective of their fitness values. Fig.2 illustrates this phenomenon. Surprisingly, it seems that in the absence of selection, the

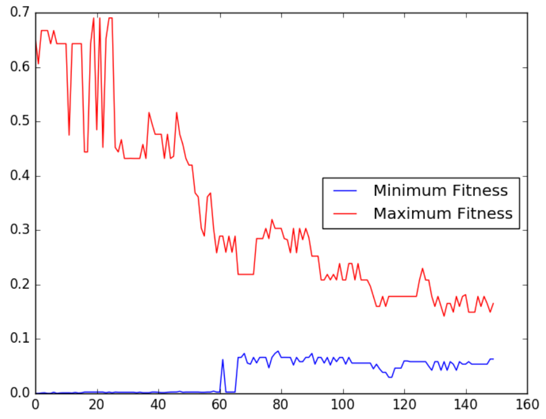

Figure 2: Effect of no selection on the evolution of individuals

population somehow evolves to contain individuals with much less fitness than what we start with. This plot was obtained for the similar settings as the plot in Fig.1, even then just decrementing the tournament size from 2 to 1 produced such a dramatic effect. A possible explanation of this observation is that in the absence of selection, the only governing factors for evolution are crossover and mutation. Since the mutation is kept almost negligible (as will be explained shortly), crossover plays a major role in deciding this behavior. The low fitness individuals when crossed with high fitness individuals seem to produce offsprings with not so high fitness values. This sounds about right since randomness is exactly about not being able to characterize using patters. If individuals could be described using schemas (i.e. they have less randomness in them), then crossing them with the ones that do not have a good schema representation will introduce non-randomness in the form of these schemas in the latter, hence, reducing the fitness values of the offsprings. The net effect is that the low fitness individuals win the race and the population contains less fit individuals as the generations pass.

3.3 Crossover

In genetic algorithms, crossover is a genetic operator used to vary the programming of a chromosome or chromosomes from one generation to the next. It is analogous to reproduction and biological crossover, upon which genetic algorithms are based. Cross over is a process of taking more than one parent solutions and producing a child solution from them. The variant of crossover that I use in this experiment is called one-point crossover, in which a single crossover point on both parents' organism strings is selected. All data beyond that point in either organism string is swapped between the two parent organisms. It is surprising that such a simple crossover function is able to produce the desired results. However, the algorithm is sensitive to the cross over probability, that is the probability that two mating individuals will actually undergo crossover. It seems that the higher the crossover rate, the faster is the convergence of minimum fitness to the maximum fitness, which means the faster our population evolves to a homogenous collection of high fitness individuals. As this probability is reduced, this convergence becomes slower. Plots in Fig.3-6 illustrate this effect. However, the plots also tell us even low crossover rates lead to evolution of good fitness individuals and hence, crossover is one of the crucial operations in this experiment.

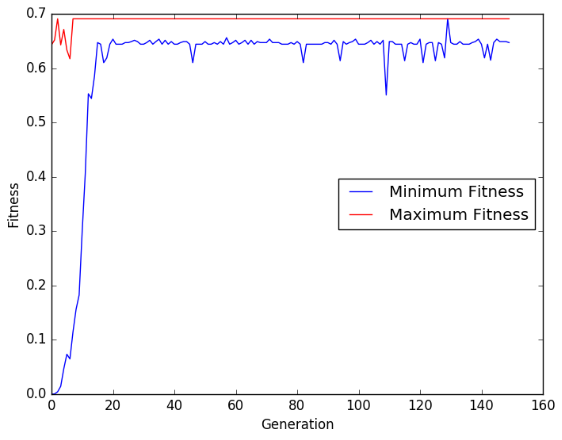

Figure 3: Evolution with crossover probability = 1.

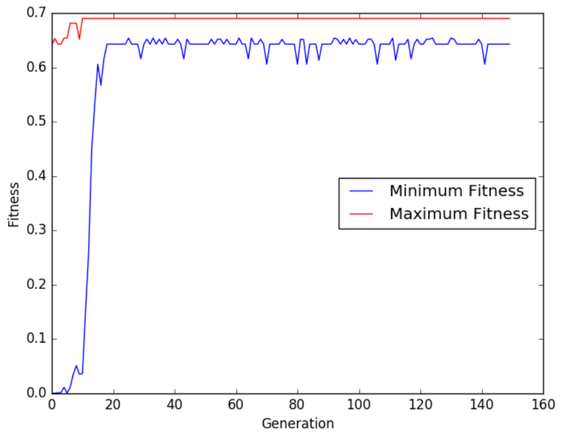

Figure 4: Evolution with crossover probability = 0.7

3.4 Mutation

Mutation is a genetic operator used to maintain genetic diversity from one generation of a population of genetic algorithm chromosomes to the next. It is analogous to biological mutation. Mutation alters one or more gene values in a chromosome from its initial state. In mutation, the solution may change entirely from the previous solution. Hence GA can come to better solution by using mutation. Mutation occurs during evolution according to a user-definable mutation probability. This probability should be set low. If it is set too high, the search will turn into a primitive random search. In our case, this probability is set to be very low, sometimes as low as 0.01%. Fig.7-11 show plots for the effect of mutation probability on evolution.

The plots show how extremely sensitive the genetic algorithm is to the mutation probability and also the fact that we are way better off with negligible mutation. Even tolerance to 0.1% mutation probability is not enough. This highly suggests that we must keep mutation rates to be extremely low or maybe even zero if we want our genetic algorithm to evolve individuals with high fitness values. It is very difficult and counterintuitive to explain this sensitivity. On one hand, we are trying to evolve random sequences to be more random, which would suggest that mutation should be a life saver in such a scenario. But on the other hand, mutation is taking away some aspect of the random sequences which is dramatically decreasing their

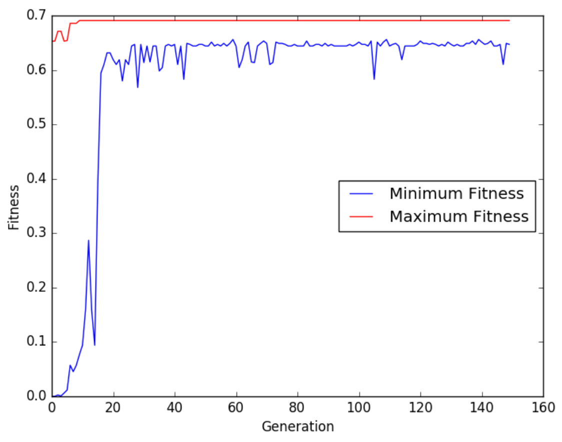

Figure 5: Evolution with crossover probability = 0.4

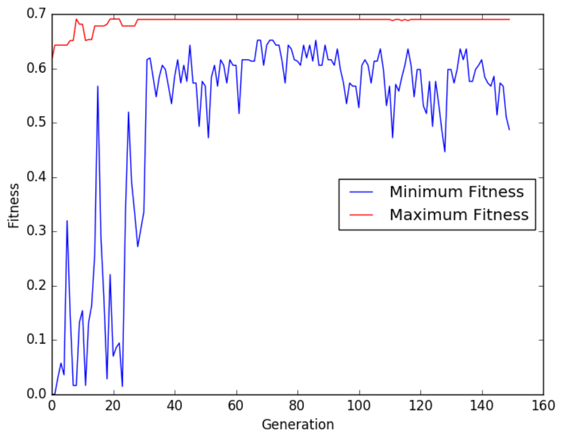

Figure 6: Evolution with crossover probability = 0.1

fitness values. All this seems to suggest that size of neutral regions in the space of random sequences is perhaps limited to only 1, since even single point mutations have the potential to decrease the fitness value dramatically.

This phenomenon is the main reason the author chose this experiment for the extra credit for this class and would love to explore the topic in detail.

4 Discussion and Conclusion

To conclude, this report presents an experiment which is an attempt to decrease the dependence of deterministic state-of-the-art random number generators like the Mersenne twistors on their seed value through evolution of the random sequences produced by them using a genetic algorithm. The results suggest that a high crossover rate and negligible mutation evolves an initial population into one that contains a homogenous population of high fitness individuals. However, the extreme sensitivity of this evolution to mutation probability is a subject of both interest and amazement to the author, which he hopes to explore in further detail.

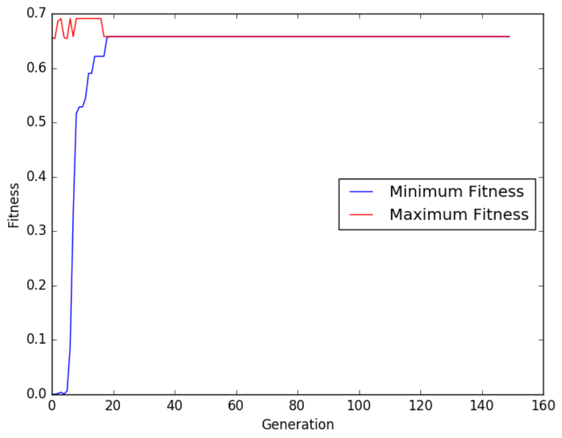

Figure 7: Evolution with no mutation.

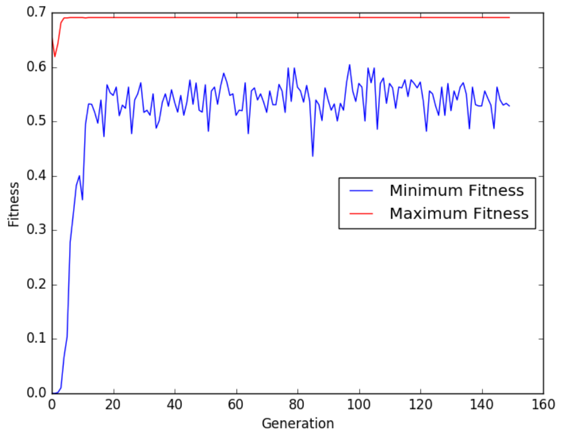

Figure 8: Evolution with mutation probability = 0.001

References

[1] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman. Mining your ps and qs: Detection of widespread weak keys in network devices. In *USENIX Security Symposium*, pages 205–220, 2012.

[2] E. Jones, T. Oliphant, and P. Peterson. {SciPy}: Open source scientific tools for {Python}. 2014.

[3] D. E. Knuth. The art of computer programming, vol. 2. *Seminumerical Algorithms*, 1981.

[4] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[5] M. Saito and M. Matsumoto. Simd-oriented fast mersenne twister: a 128-bit pseudorandom number generator. In *Monte Carlo and Quasi-Monte Carlo Methods 2006*, pages 607–622. Springer, 2008.

[6] L. M. Schmitt. Theory of genetic algorithms. *Theoretical Computer Science*, 259(1):1–61, 2001.

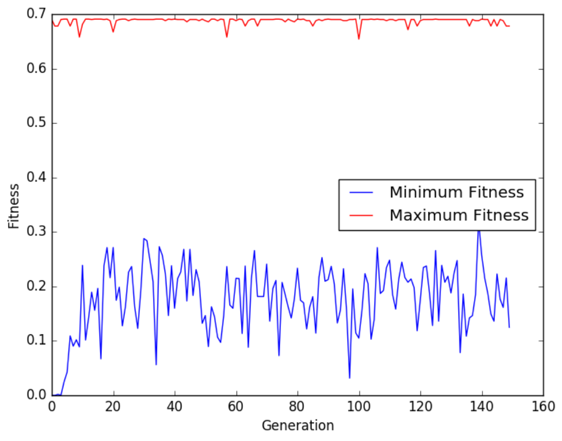

Figure 9: Evolution with mutation probability = 0.01

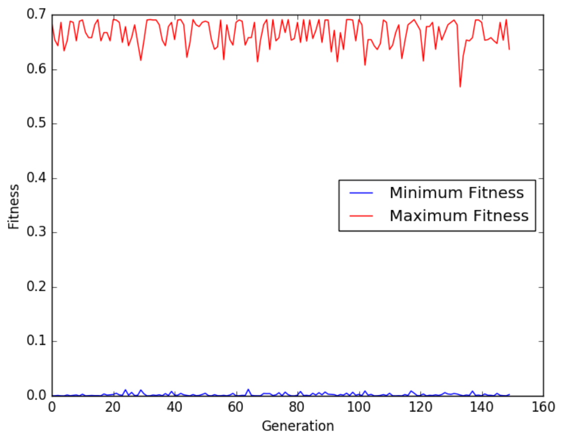

Figure 10: Evolution with mutation probability = 0.1

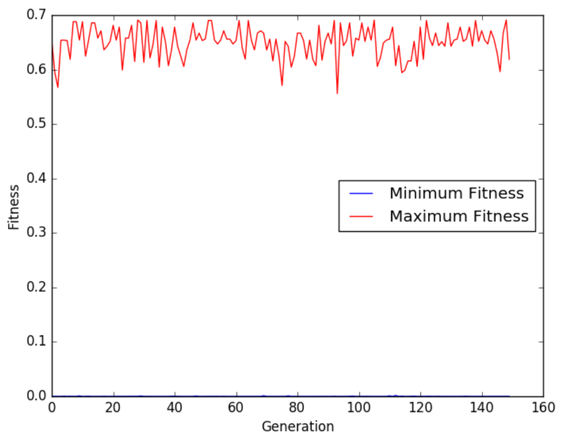

Figure 11: Evolution with mutation probability = 0.3