**The Effects of Parameter Changes on Comparable Evolutionary Algorithms**

22019799

# Introduction

Set out to find the optimal configuration for an Evolutionary Algorithm (EA), the experiment attempts to understand each argument and its resultant effect on the solution of the fitness. A well-reasoned thought might suggest trying random numbers, and extrapolating tests from initial findings. That is not an efficient solution, nor is it effective. Instead, this experiment focuses on an underlying Meta program which can incrementally test every value in a range, presenting the best results.

The same form of experimentation will be carried out on two different minimisation equations. With the project’s backbone consisting entirely of C++, it would be able to store all the necessary data in relevant object types. This ensures that all tests can be independently explored and processed accordingly.

These results are then plotted graphs using an open-source GUI library, allowing them to be further analysed by an operator. This process helps to develop a deeper understanding on the effects of the parameter changes.

A small portion of the pattern can be observed by first recognising that the solution’s fitness increases with the tournament size. The opposite could be said for both Mutation Rate and the Mutation Height: as these numbers grow, the solution’s fitness shrinks dramatically.

# Experimentation

Evolutionary Algorithms typically begin with a pre-defined set of data to manipulate. Given that the experiment is entirely focused on the effects of the parameters belonging to the EA, using pseudo random genes would be equally acceptable. To determine if an individual’s genes hold any merit: a fitness scoring function must be used.

After scoring the whole population, next step is to feed these individuals into a Tournament Selection process. The basis for this comes from the most known examples of the Roman Empire. Gladiator fights were common practice and served a way for spectators to gather and watch as two warriors fought to the best of their abilities. This method of fighting gave way to a form of forced natural selection, exemplified by the first stage of the EA. Initially, the algorithm should loop through the population, choosing a champion each time and selecting the other challengers from the population at random. In each gladiator battle, only one can be the victor. This is determined by checking who, of the current gladiators, has the highest fitness score. The victor is then passed onto a new list of individuals, moving the cycle onto the next champion in the population.

Once the list of champions is complete, the Crossover function is initiated. As the name may suggest, it draws inspiration from the molecular biology method of DNA modification known as Gene Splicing. In a rough outline, this technique involves cutting a piece of the subject’s DNA from one enzyme and replacing it with an identical cutout from another, more admirable enzyme. A similar process is attached to this function. Unlike the Gladiator battle, this function depends on visiting each individual exactly once. To accomplish this, the population loops through every other member and selects the next member by simply indexing the next individual in the population. Here, a random number between zero and the number of defined genes is generated. This number is then treated as a point in the gene list, after which the remaining genes are transferred to the other individual. Both individuals have their fitness recalculated, and then are transferred to a new population list.

Finalising the Crossover function is imperative for the Mutation Sequence to begin. Once again using biology as a backbone, more specifically the fields of virology and evolutionary biology, mutation serves to introduce chaos, or chance, into an algorithm. Inside the function, each individual is once again looped, and so are all of their genes. For each gene, a random first number is generated. If this number is lower than a predefined Mutation Rate, a second random number is generated. The second number will be between the positive and negative values of the Mutation Height, another predefined entity. This signed variable is then added to the gene. However, if this addition moves the gene beyond the accepted upper, or lower, thresholds: the gene is instead replaced with the accepted threshold. After the genes have all had the chance to mutate, the fitness is recalculated for a final time, and the population is copied to a new list.

On completion of the population alterations, only one step remains. Evolution is the driving field behind this type of search algorithm, and focuses on the contributions made by Darwinism. In natural selection, the individuals with the best features are more likely to survive. Over generations, this repeated form of natural selection adapts the individual to fit their environment. The exact same process can be forced onto an EA, this is the method called Elitism. In the realm of Artificial Intelligence, this selection process involves surveying each population’s fitness mean average. The most favourable individuals are passed through to the next generation: restarting the cycle.

Different strengths of elitism could be applied, a simple pass-down would ensure that all alterations are passed on to the next generation, disregarding the need for multiple populations. However, this holds potential to result in backsliding, decreasing the population’s average fitness over time. More sophisticated options would see each population tested. This, too, would be a mistake. Instead of incorporating mutations every time the population advances, they would only be included if the mutated population scores higher than the others.

Since Mutations have a 50% chance of being a negation from the altered gene, this also means that: because the fitness could remain the same, the chance of the mutations being carried over is even less. That leaves one last option: only the final population is weighted against the initial population. The victor proceeds. This ensures that if any progress has been made at all, the mutations created will also be passed down, ensuring the algorithm never hits a brick wall.

As previously stated, plugging random numbers into the positions of mutation height, mutation rate, and tournament size is not optimal. It would take too long, and automating the process is far more efficient. Three nested loops run, incorporating the described Evolutionary Algorithm. At the end of each test, a data structure is created which includes: the current test’s parameters, the ending fitness average, and the solution’s fitness rating. The solution’s fitness rating is calculated by multiplying the final ending fitness by the number of loops performed.

The final piece of data stored in the new structure is the generation at which the increase in fitness stops changing or plateaus. This is crucial for calculating the overall fitness of the solution, because a lower generational plateau would indicate a solution is found quicker. Ensuring the parameter search finds the most optimal solution is vital; without factoring in the plateau, this is not possible.

Once the meta-data list of tests has concluded, graphs are available for viewing. The combination of these different graphs allow for an easy understanding of the cause-effect relationship existing between the arguments inserted to the algorithm, and the efficiency of the results. A key detail in the determination of an argument’s outcome, however, is also the fitness calculation used. Two minimisation equations were used to compare results which are applicable for general, and specific models.

Knowing how these nested loops are structured is one of the most important things in understanding the graphs’ presentation. On the outside, there is a loop testing the Tournament Size (Represented by %S) by checking every value between the current systematic search bounds. The second loop is structured to test the bounds of the Mutation Rate (Represented by %MR), followed by a loop exploring the Mutation Height (Represented by %MH) bounds.

The loops are simple. Each time, they run the Evolutionary Algorithm; arguments increase incrementally. In each test, the Tournament Size is increased by one with each iteration. Both the Mutation Rate and Mutation Height are consistently increased by 0.01.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Operator Variable | Large Test | Medium Test | Small Test | Inspection |
| Tournament Size (%S) | 2 -> 10 | 2 -> 10 | 2 -> 10 | 3 -> 10 |
| Mutation Rate (%MR) | 0.01 -> 0.5 | 0.01 -> 0.3 | 0.06 -> 0.3 | 0.02 -> 0.2 |
| Mutation Height (%MH) | 0.01 -> 0.5 | 0.01 -> 0.3 | 0.06 -> 0.3 | 0.06 -> 0.2 |

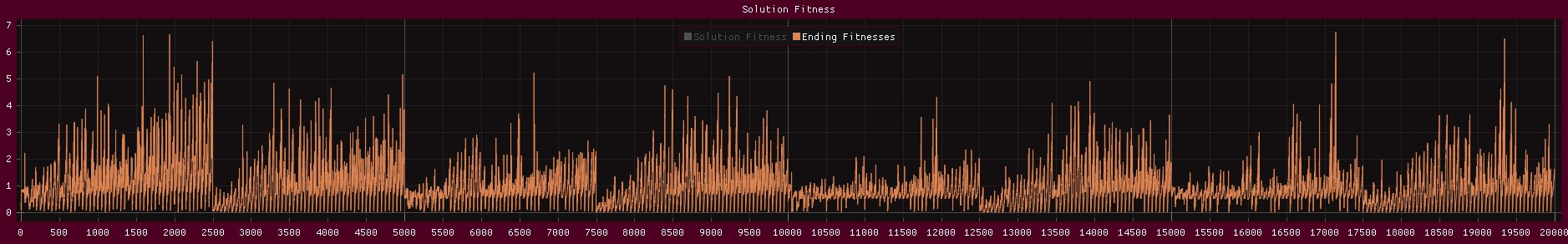
Before the graphs can be explained in their entirety, the gene fitness calculations must be defined. A couple of math equations

Description automatically generated with medium confidence

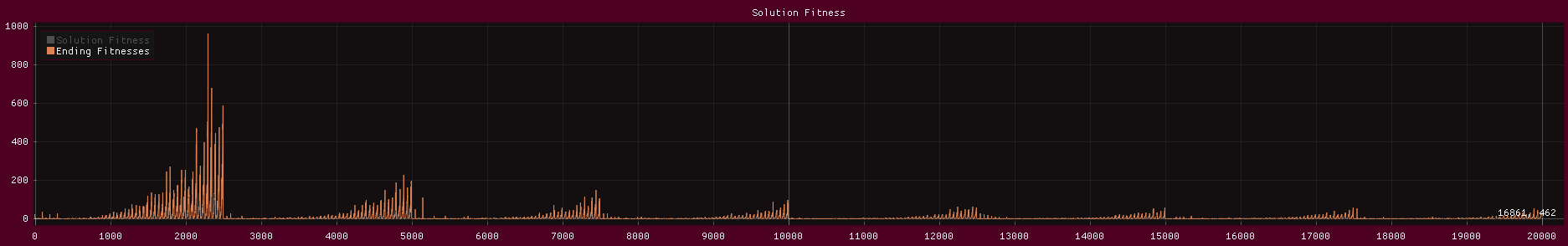
(Article 0 – Minimisation functions one and two)

These two functions are vastly different and have different advantages to both. The real power, however, is combing the output of them both and cross referencing the best results. While minimisation function one appears to be the most simplistic, it is the most inefficient for computing power. This must be factored in when trying to select a model to use, an inefficient model would cause the algorithm to draw in much more computing power than necessary. Over 100 tests, the first equation completes 100 generations and averaged 0.0186s. The second equation averages 0.0135s.

Though this difference may seem inconsequential, its important to cascade the buffer’s impact when servicing potentially thousands of requests. This buffer’s primary cause is the difference in the position of the squared number in the equation. Instead of only having one inside a loop, there are two. Simply speaking, this near doubles the number of calculations needed because of how computers handle mathematics.



(Article 1- Graph of Ending Fitness of Minimisation One – Large Study)

(Article 2 - Graph of Ending Fitness of Minimisation Two – Large Study)

As previously discussed, there are some things which are argument dependant. Others are minimisation dependant. A resultant can easily be determined as being controlled by the minimisation function by cross referencing the function’s graphs. If similarities are established, they may be argument dependant. Otherwise, they are most likely to be controlled by the fitness function. Clear evidence of a case, such as this, is that: whilst in Article 1 there is an obvious relationship between the tournament size being even, and the consistency of the ending fitness. This relationship is not apparent with the second function.

On the opposite side of the spectrum, the first thing that should be noticed as a simple cause-effect relationship: whilst the tournament size increases, the ending fitness average decreases. Although this is true, it is important to note the drawback. Increasing the tournament size exponentially increases the number of loops needed to complete the tournament. If this subsequently passes a threshold; the solution may reach a better ending fitness, but it would demand too many computing resources. As the tournament size increases, obviously so do the number of randomly selected Challengers in the tournament. This does also mean that the chance for an individual to be duplicated by winning more than one tournament increases, allowing for a quicker repopulation of higher value individuals.

Initial findings would suggest that striking the lowest tournament selection options could be a good idea. This would, in turn, remove the higher spikes from the parameter search. It is important to first note the reason for the spikes in the ending fitness average.

A screen shot of a computer

Description automatically generated

(Article 3 – Zoomed in Article 1, **Generations** 0 -> 2500, **%S** = *2*)

A screen shot of a game

Description automatically generated

(Article 4 - Zoomed in Article 2, **Generations** 0 -> 2500, **%S** = *2*)

A graph on a black background

Description automatically generated

(Article 5 – Zoomed in Article 2, **Generations** 0 -> 600, **%MR** *0.01 -> 0.12*)

Leading onto the second relationship established: when the Mutation Rate is increased, the ending fitness height becomes less desirable. The rise of the Mutation Rate means that the gene pool holds less integrity, and eventually becomes unstable. Eventually this results in the complete loss of admirable genes, as represented by Article 3 and 4. While this does consistently reduce the time taken for the algorithm to plateau, the ending result is significantly worse.

A graph on a screen

Description automatically generated

(Article 6 – Zoomed in Article 1, **Generations** *300 -> 350*, **%MR** = *0.07*, **%MH** *0.01 -> 0.5*)A screen shot of a graph

Description automatically generated

(Article 7 – Zoomed in Article 2, **Generations** *300 -> 350*, **%MR** = *0.07*, **%MH** *0.01 -> 0.5*)

Like Mutation Rate, Mutation Height has a simple cause-effect relationship. When the height rises, the fitness once again becomes less desirable. The same precedent holds true; although in this case, the loss of integrity is much steeper. Ultimately, this results in the ending fitness being exponentially high, especially when combined with a higher Mutation Rate. Using both, a high Mutation Hight and Mutation Rate, would cause a phenomenal reduction in how good the output is. For example, when **%MR** and **%MH** both equal 0.5: roughly every other gene is being altered. This could be altered by anywhere between -0.5 and +0.5.

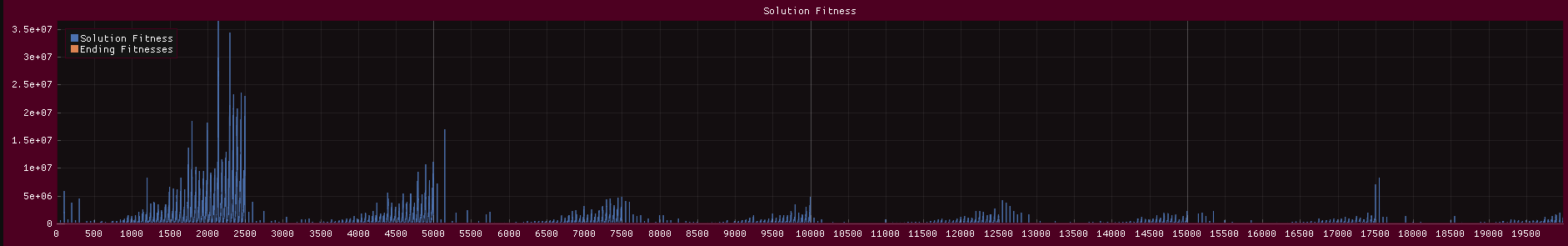
The synergy of a large Mutation Height and Mutation Rate have already been exposed. Similar reactions occur when faced with the combination of a low value for each. Opposite to the complete loos of integrity, the gene pool now holds too many original values. Either taking too many generations to reach a plateau, or not finding one at all; this can result in not finding a desirable outcome, causing the algorithm to plateau at the original lowest value.

Discussing the difference between a good solution and an optimal solution is only possible once the efficiency is first calculated. There have been reviews of the negative effects of incorporating specific parameters. Using this knowledge, it is possible to calculate a formula which totals the number of loops preformed by the Evolutionary Algorithm, and multiply it by the ending fitness. This would cause any fitness below 1 to act as a divisor, reducing the total solution fitness rating. The opposite would happen when the ending fitness rests above 1: pushing a bigger gap between the optimal and poor solutions. That would, in turn, force the most optimal solutions to present themselves by factoring in computing resources needed to find an equal ending fitness.

A screen shot of a graph

Description automatically generated

(Article 8 – Graph of Solution Fitness Rating for Minimisation One – Large Study)



(Article 9 - Graph of Solution Fitness Rating for Minimisation Two – Large Study)

As discussed, the spikes apparent on Articles 8 and 9 are a clear indication of times where the solution either didn’t plateau, or the solution did not reach a desirable result. Using the generational inspection tool, its easy to see which parameters should be clamped in the meta search. The biggest cause for the higher spikes: the Mutation Rate and Height are over 0.3.

A screen shot of a graph

Description automatically generated

(Article 10 – Graph of Solution Fitness Rating for Minimisation One – Medium Study)

A screen shot of a computer

Description automatically generated

(Article 11 – Graph of Solution Fitness Rating for Minimation Two - Medium Study)

As evidenced by the stark contrast between the two studies, removing the higher spectrum arguments quickly gives a multitude of more optimal solutions and keeps the results within much closer boundaries. Shown by the remaining spikes and previous statements, something else that will regularly decrease the optimisation of a solution: the lower Mutation Rate and Height combination.

Small alternations to the next round of testing, include altering the minimum limit for the mutation height. This takes advantage by retaining optimal solutions, while discarding the insignificant.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| %S | %MR | %MH | Ending Fitness | Plateau | Solution Fitness Rating |
| 9 | 0.06 | 0.02 | 0.002006 | 69 | 214.559723 |
| 9 | 0.18 | 0.01 | 0.002955 | 84 | 384.724152 |
| 7 | 0.11 | 0.02 | 0.003436 | 76 | 404.759644 |
| 9 | 0.28 | 0.01 | 0.004092 | 63 | 399.564514 |
| 7 | 0.26 | 0.01 | 0.004227 | 74 | 484.875610 |

(Article 12 – Graph of top 5 solutions by ending fitness, minimisation one, medium test)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| %S | %MR | %MH | Ending Fitness | Plateau | Solution Fitness Rating |
| 9 | 0.06 | 0.02 | 0.002006 | 69 | 214.559723 |
| 7 | 0.04 | 0.03 | 0.003839 | 46 | 273.748596 |
| 7 | 0.13 | 0.01 | 0.002646 | 72 | 295.326141 |
| 7 | 0.19 | 0.01 | 0.002517 | 89 | 347.189606 |
| 9 | 0.18 | 0.01 | 0.002955 | 84 | 384.724152 |

## show the difference between the top10 for the large and medium

## explain decisions to reduce the range further

## discuss in detail the solution fitness for the tiny exam

##explain why the inspection is the most valid set of test data

## show the generational runs of the final top fitness and top solution fitness

## explain why they are the overall best

# Comparison

# Conclusions