Title: Using Rust to Make a GA-Simulation-Runner / Graph-Generator
Date: 02/28/2023

1.0 Introduction

1.2 Outline

- *Section 1.0 Introduction* — contains this outline and an abstract of this paper.
- *Section 2.0 Program Description* — provides details about how the genetic algorithm system works and describes my journey of implementation.
  - *Section 2.1 The Genetic Algorithm* — details about the GA implementation.
  - *Section 2.2 Simulation Batching* — details about how I used multithreading to run multiple simulations concurrently.
  - *Section 2.3 Graph Generation* — details about how I generated the graphs used in the results section of this paper.
  - *Section 2.4 Tying it All Together* — final implementation details.
- *Section 3.0 Results* — contains the 6 graphs that were generated using the program.
- *Section 4.0 Conclusions* — is where I discuss the results that were found after running the batch simulator with various variations on parameters.

1.1 Abstract

This paper describes a program I built in Rust using the crates genevo and plotters, which is able to run averaged batches of genetic algorithm simulations in parallel given various sets of parameters and variations to apply to one of the parameters. It then generates graphs based on the generated data, where the y-axis represents fitness (averaged over batch results for each line on the graph) and the x-axis represents the passage of generations in the simulations. I used this program to generate 6 graphs, each showing average batch fitness over generations for one or more sets of parameters:

1) Default parameters.
2) Default parameters, but with various different numbers of individuals per parents.
3) Default parameters, but with various selection ratios.
4) Default parameters, but with various mutation rates.
5) Default parameters, but with various reinsertion ratios.
6) A parameter set defined by taking the best-performing value for each of the parameters in the previous tests.

2.0 Program Description

This section provides a description of my program and all of its parts. It's split into three sections, each of which explains an aspect of how the whole program works together.

2.1 The Genetic Algorithm

After deciding to make this project my first Rust project, I started designing my own genetic algorithm in Rust. But Rust has a very strict typing system, which contributes to its steep learning curve – and I was struggling to implement the system I was envisioning – so I decided to instead look for a crate that helps with the implementation of genetic algorithms in Rust, which is what led me to genevo. To run a genetic algorithm with genevo, there are a few things that need to be defined first (implementation specific details provided in the parenthesis):

1. The population size (256).
2. The maximum number of generations to let a simulation run before stopping it (16,384, which is 2^{14}).
3. The number of individuals to generate in the crossbreeding phase of the algorithm (I decided to try various values of this parameter, and did the same for a few other parameters).
4. The selection ratio, which controls the ratio at which parents should be selected for the crossbreeding phase (varied).
5. The mutation rate, which controls how often the algorithm should randomly mutate individuals of the population (varied).
6. The reinsertion ratio, which controls how much of the population should be replaced in the reinsertion phase of the GA (varied).
7. The phenotype ("Phenome", which is a type alias for "String").
8. The genotype ("Genome", which is a type alias for a vector of "Nucleotide"s, where a nucleotide is an enum of the values **A/C/T/G**).
9. The length of a genome (100).
10. How to convert a genome to a phenome (by using "map" to iterate over the vector of nucleotides, matching each enum value to its letter as a character, and collecting the resulting vector of characters into a string).
11. How to generate random solutions (by using a standard distribution to choose randomly between **A/C/T/G**).
12. How to randomly mutate solutions (using the same method as above).
13. How to calculate:
    a. The fitness of a given genome (I decided to define the fitness of an individual as the number of 4-long chunks – using a fixed frame – that contain only one variation of nucleotides. To implement the calculation of this fitness, we first create a mutable variable to store the number of chunks that match the criteria, then we divide the genome's string into chunks of size 4 and iterate over the

chunks, checking that the 2nd, 3rd, and 4th nucleotides match the 1st. If they all match, we increment the variable. At the end, we return the variable.).

b. The average fitness of a list of genomes (generic idiomatic implementation).
c. And finally, the highest and lowest possible fitnesses (Highest: 25, because there are 25 4-long chunks in 100, which is the size of a full strand of DNA. Lowest: 0, which would be the case in which every chunk is heterogeneous).

While developing this project, I realized I wanted to be able to run different parameter sets during a single execution of the program. This led to the struct named “Parameters”, which contains fields for the name of the parameter set and all of the parameters above that were listed as “varied”.

The “Variation” enum is used to store a list of variations over a single parameter in a parameter set. It can take the values **Default**, **NumIndiv**, **Selection**, **Mutation**, **Reinsertion**, and **BestOfEach** (these correspond to the graphs that were generated). If the enum takes any of the values except **Default** and **BestOfEach** (each of which should only contain a single set of parameters), the enum stores not only the type of variation, but also a Vector of the appropriate type to vary the correct parameter.

The “run_sim_from_parms” function takes an instance of “Parameters”, using it to run a single simulation, and returns either “None” if there was an error during the simulation or the simulation ran to the generation limit before finding a perfect solution, or returns a “DataSetWithLabels” if a perfect solution was found. As I’ll describe in the next section, I used many calls to run_sim_from_parms in different threads to generate data from many simulations at once.

2.2 Simulation Batching

Following the idea of running multiple parameter sets in one execution, I decided I wanted to run simulations in a multi-threaded way, and also wanted to be able to run multiple simulations for each parameter set to get data that is averaged over multiple runs.

The function called “run_sim_batch” takes a list of parameter sets and returns a DataSetWithLabels (which represents a final, averaged data set). It does so by spawning 16 threads for each parameter set in the list, each of which returns a vector that holds that simulations best found fitness score over the generations. It uses two Rust constructs (Arc and Mutex) for memory safety over threads. While joining all the threads, data is collected into lists (for each of the parameter sets) of lists (for each of the 16 simulations that were run), then averaged (within each parameter set, over each of the 16 sims) and named to get the final data set.

2.3 Graph Generation

To generate graphs in Rust, I used a crate called plotters. The “generate_graph” function accepts a DataSetWithLabels and a string (for the name of the PNG file to be generated), and returns nothing. It contains the needed code to generate graphs with plotters, and iterates over the data set, drawing a labeled line for each parameter set that was simulated with the batch simulation system.

The final piece of the puzzle is the “generate_graph_from_variation” function, which is basically a wrapper function for generate_graph that first checks whether it should generate the output file name that was passed, then calls generate_graph if so.

2.4 Tying it All Together

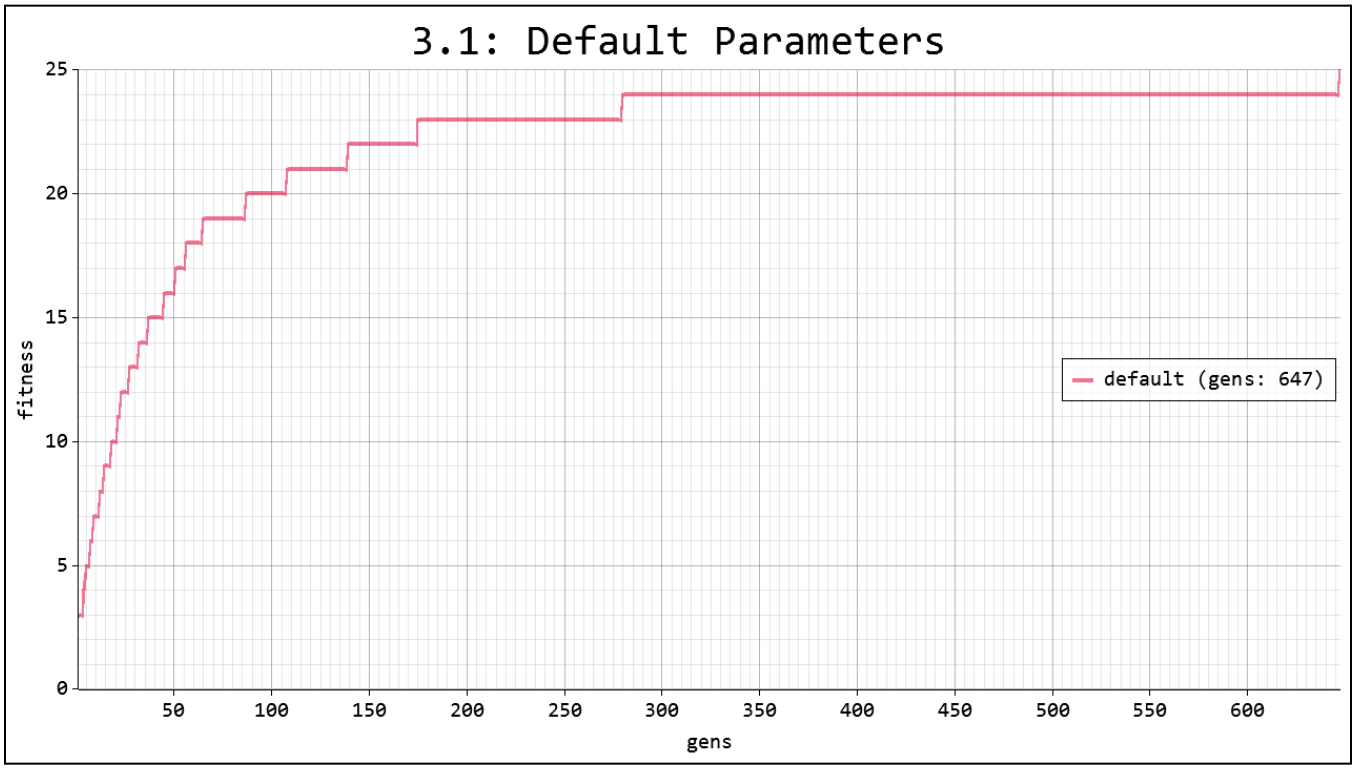
In the “main” function of this program, firstly an assertion is made that will crash the program if the strand size is not divisible by 4, which is required by the chosen fitness function. Then, the files that will be overwritten are first deleted, then finally, calls are made to generate_graph_from_variation to generate all the graphs that are marked to be generated.

3.0 Results

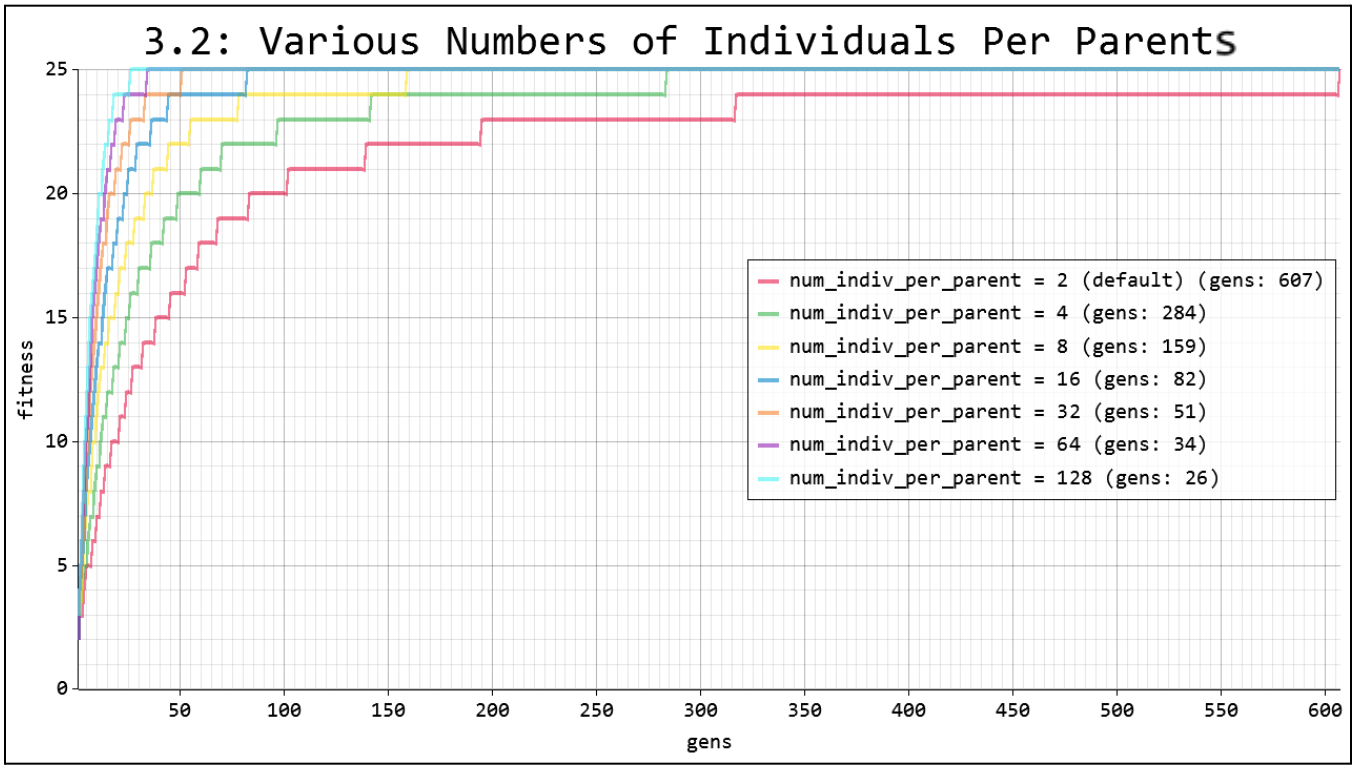
This section contains the graphs that were generated by my program using the plotters crate (image generation details in Section 2.3). As explained in Section 2.2, the graphs are made using data that was generated by: running a batch of 16 simulations for each of the parameter sets to be tested, then, for each simulation batch, averaging each simulations’ best found fitness at each generation to produce the final lines that represents the performances of a each of the parameter sets. The x-axis represents the amount of generations passed in a simulation batch and the y-axis represents the average of the best fitnesses found in each of the simulations in the batch. When a line reaches the top of the graph, it means that all of the simulations in that parameter set’s simulation batch have found at least one solution with a perfect fitness. The width of the graph represents the maximum number of generations run by any of the simulations for any of the parameter sets on that graph.

Graphs 3.1 and 3.6 each contain a single parameter set, while graphs 3.2-3.5 each contain multiple parameter sets. Graph 3.1 uses a *guesstimated-by-me* “default value” for each parameter (num_individuals_per_parents: 2, selection_ratio: 0.5, mutation_rate: 0.05, reinsertion_ratio: 0.5). Graphs 3.2-3.5 each use the default parameters, but with multiple (5-7) variations to one of the parameters, which is known by the title of the graph. Finally, Graph 3.6 uses the fastest-learning values found for each parameter.

3.0 Results (continued)

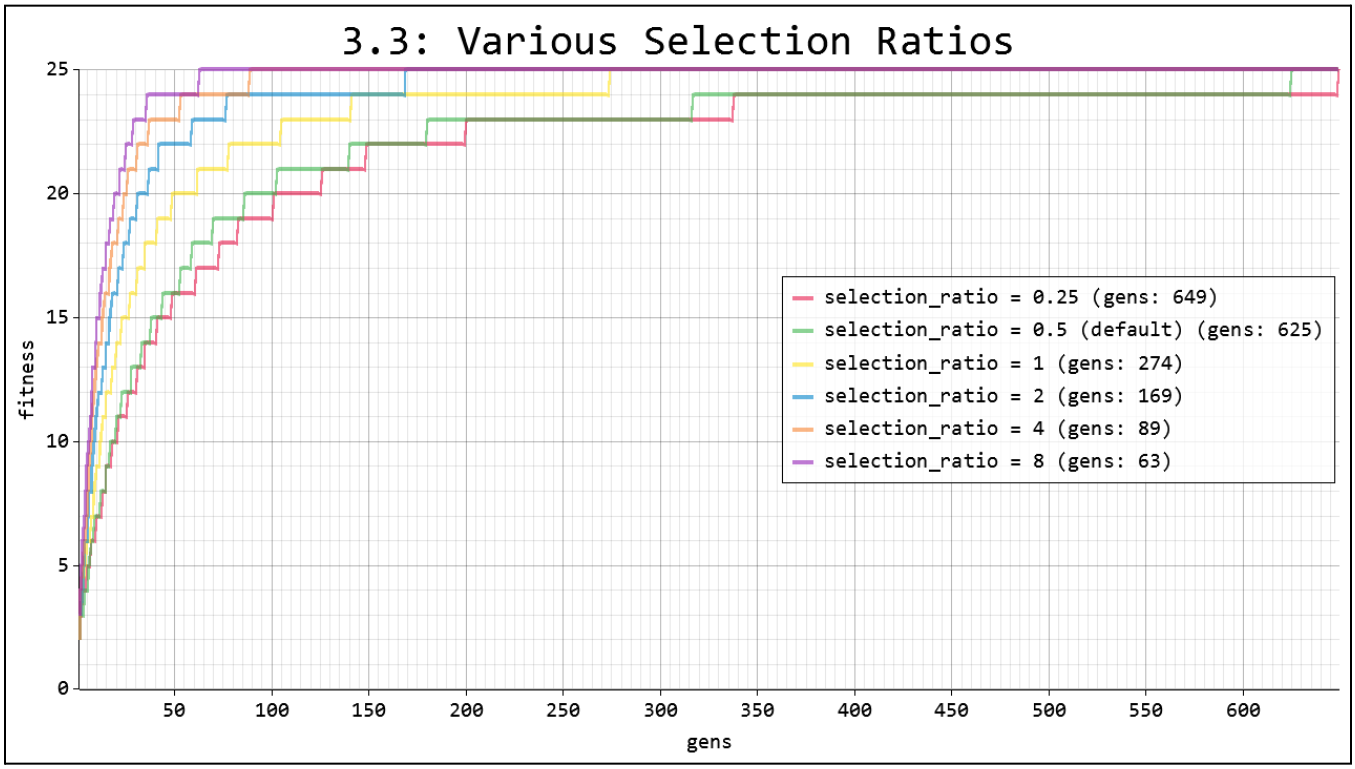


Jump to Section 4.1

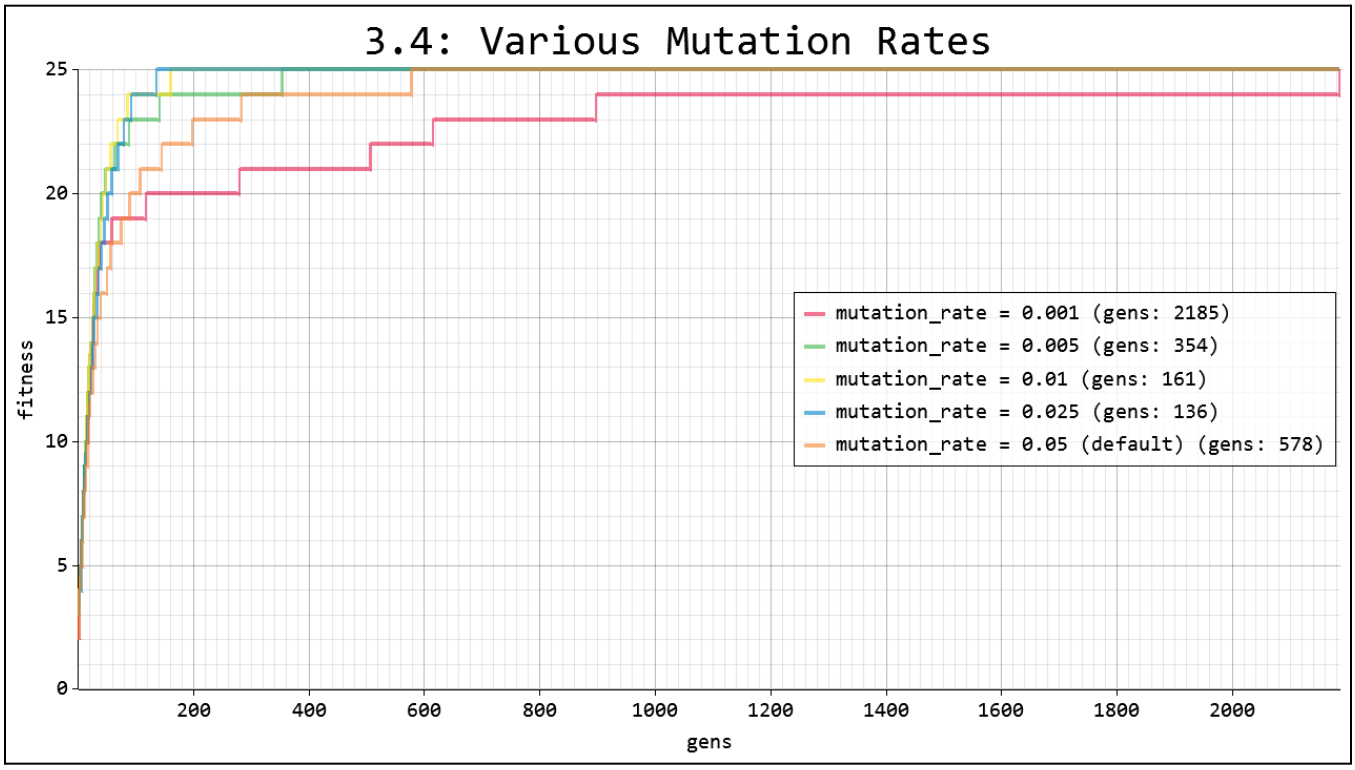


Jump to Section 4.2

3.0 Results (continued)

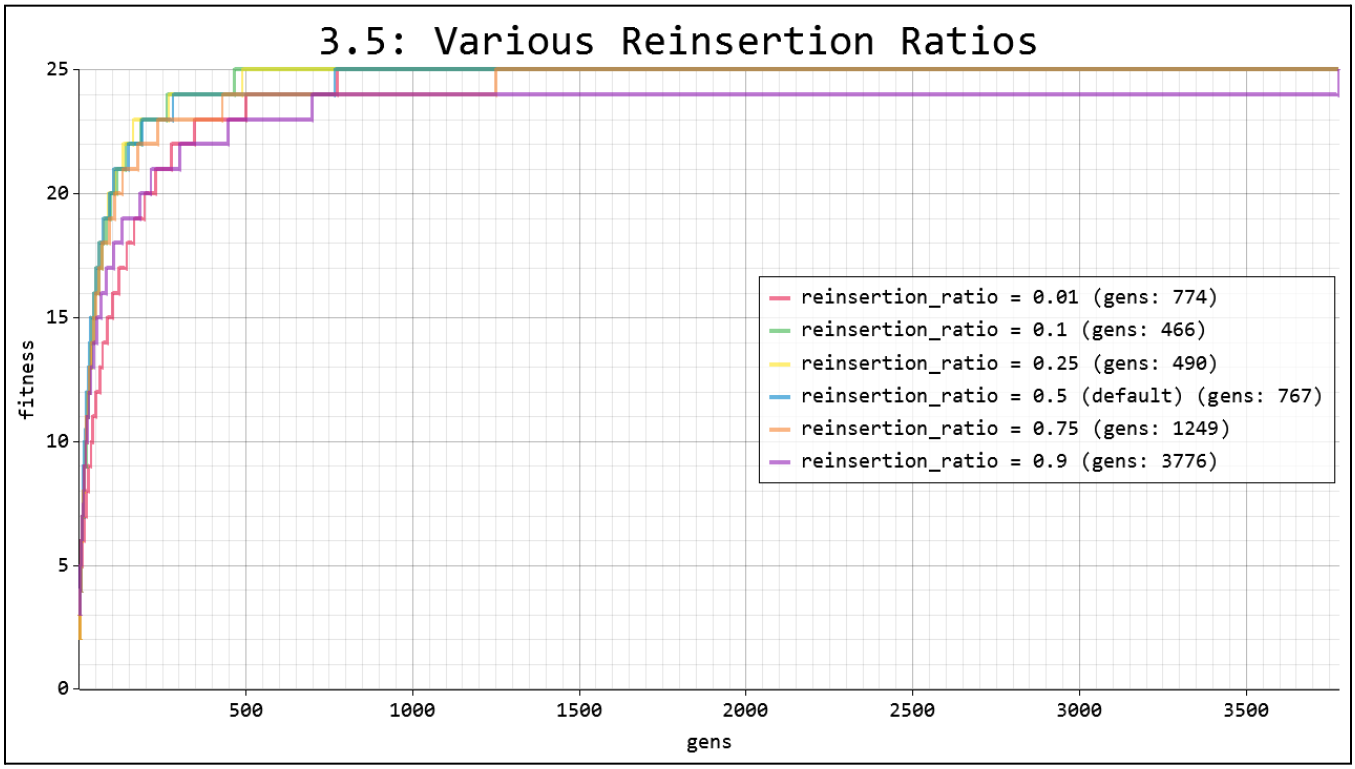


Jump to Section 4.3

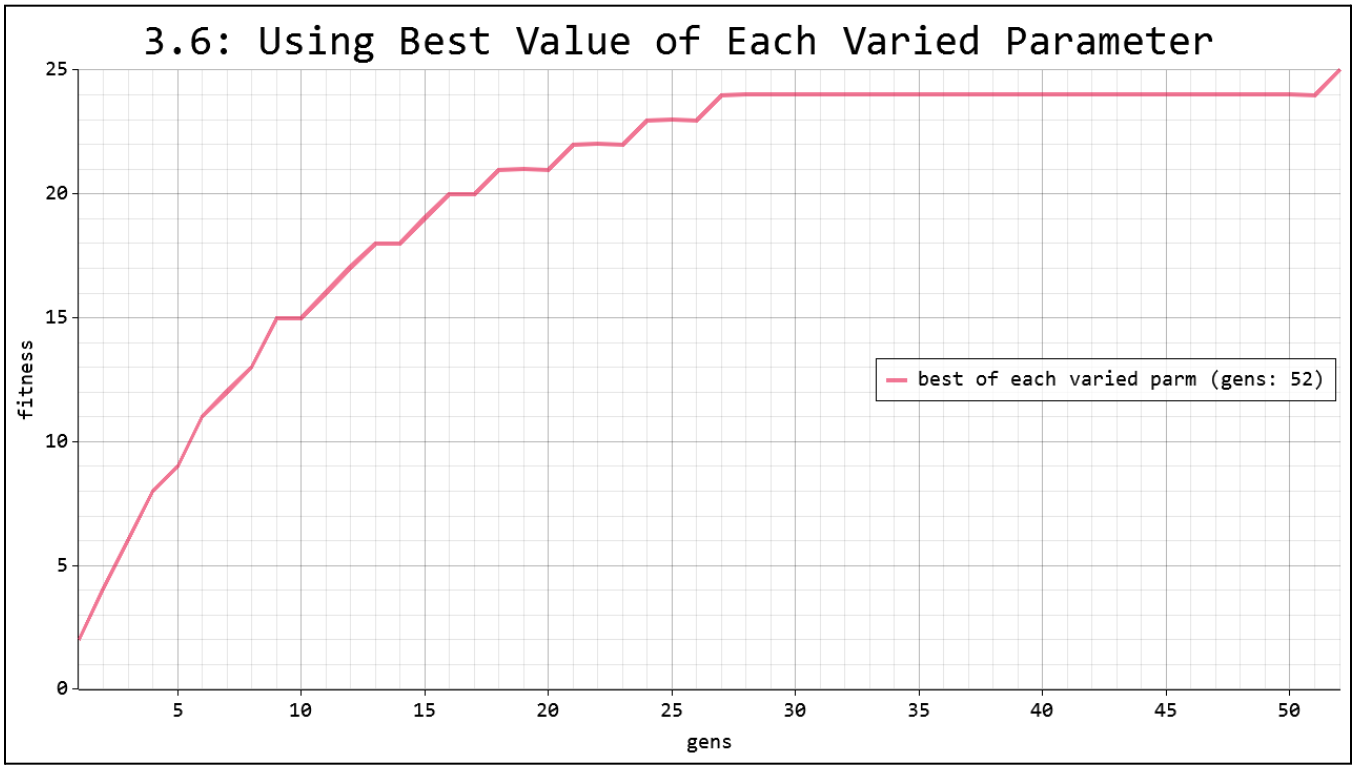


Jump to Section 4.4

3.0 Results (continued)



Jump to Section 4.5



Jump to Section 4.6

4.0 Conclusions

This section contains conclusions about the graphs in Section 3.0. The titles are links to the graphs.

4.1 Default Parameters

This graph shows that using default parameters (num_individuals_per_parents: 2, selection_ratio: 0.5, mutation_rate: 0.05, reinsertion_ratio: 0.5), most of the simulations found a perfect solution by generation 280, and out of the 16 simulations, the one that took the longest took 647 generations.

4.2 Various Numbers of Individuals Per Parents

Interestingly, this graph shows that the more individuals generated per set of parents, the less generations it takes the simulation to find a perfect solution. At first this confused me, but I've since realized that it actually makes sense: if more individuals are generated, then there are more individuals from which the maximizing selector to choose during the reinsertion phase. So naturally, choosing the best individuals from a larger pool of offspring will lead to better draws on average.

4.3 Various Selection Ratios

This graph shows something similar to the previous graph, but about the selection ratio. In this case, we are inflating odds again, but sooner, because the selection ratio controls the amount of potential parents to choose from.

4.4 Various Mutation Rates

This might be the graph that is most interesting to me personally, as it reveals that using a mutation rate of around 0.025 tends to be a bit more efficient (perfect solutions found in all 16 simulations in 136 gens) than using either a smaller mutation rate (0.01 -> 161 gens, 0.005 -> 354 gens) or a larger mutation rate (0.5 -> 578 gens).

4.5 Various Reinsertion Ratios

This graph is interesting for the same reason as the previous, revealing that the best reinsertion ratio is around 0.1, in which all 16 perfect solutions were found in 466 generations.

4.6 Using Best Value of Each Varied Parameter

Comparing the first graph with the final graph reveals the increase in performance achieved by using the best value for each of the parameters that were tested (default parameters: all 16 perfect solutions found in 647 generations; final: 52 generations).