
Homework N°2

Genetic Optimization Algorithm Application on the N-Queen Problem

Lecture Professor: Alexandre Bergel
Assistant Professor: Juan-Pablo Silva
Assistants: Alonso Reyes Feris
Gabriel Chandía G.
Alumni: Diego Ortego
RUT: 19.671.411-1
Github: Gedoix
Project: [link to github](#)
Date: December 2, 2018

1 Introduction

This homework's goal is to achieve an extensible object-oriented implementation of a genetic optimization algorithm using a tournament selection function, and allowing for an arbitrary genetic alphabet, evaluation function, amount of individuals per generation, amount of survivors per generation, and percentage probability of mutation for each gene during the reproduction phase.

The project was developed using [Jetbrain's Pycharm IDE](#) and built around instructions and guidelines given by the course's lectures.

The project can be found in [this github repository](#).

The necessary packages, installation instructions, and other information regarding tests can be found in the project's `README.md` file.

All files concerning this document are inside the project's `src/genetic_algorithms` package, and plotted results can be found in the `plots` directory.

2 Methods

2.1 Problems to Solve

2.1.1 Word Guessing Demonstration

The first problem to address, as a demonstration of the algorithm's effectiveness, is a simple example of arbitrary supervised word guessing.

The program should generate a random word based on an arbitrary alphabet, and of arbitrary length, and iterate a genetic learning algorithm until it can demonstrate the capacity to guess said word.

The evaluation function handed to the algorithm can be as simple as a character by character sum of errors between each individual gene chain and the expected result.

2.1.2 The N-Queen Problem

The second problem to address is a more advanced example of optimization where the space of possible solutions can quickly scale exponentially, a generalization of the https://en.wikipedia.org/wiki/Eight_queens_puzzle 8-Queen Problem

The goal is for the algorithm to be able to find one of the possible configurations of a $N \times N$ chess board with N queen pieces placed on it, such that none of them can attack each other, following the classic rules of attack in the game.

There currently is no efficient algorithm for finding the amount of this configurations for an arbitrary value of N , but there are many efficient (some of them $O(1)$) algorithms of finding one configuration for any N .

Still, the goal is for the genetic algorithm to find a solution without any external examples that would benefit pattern recognition.

.

2.2 Solution Scheme

With the goal of building a genetic general algorithm that could manage both of these tasks easily, stay extensible and well encapsulated, the scheme is follows:

- The genetic guesser needs a class of it's own
- The constructor, or builder, method of said class must allow the guesser to receive arbitrary specifications on how to generate populations, evaluate them, reproduce them, and what amount of survivors must be kept after each iteration
- The guesser should be able to answer basic performance-related questions about itself during execution, so that these values can be visualized and compared
- For testing and reproducibility purposes the class should accept a seed for it's randomizer

The initialization of the guesser should setup a starting population of randomized individuals using the supplied alphabet and seed.

Lets call the guesser through the variable name **guesser** for ease of communication purposes, initialized with the parameters:

- `individuals_amount`, the amount of individuals per iteration.
- `genes_per_individual`, the amount of genes for each individual.
- `gene_alphabet`, the list of elements that count as a gene.

And the optional parameters:

- `evaluation_function`, the fitness evaluation for every individual, defaults to returning `genes_per_individual` when unspecified
- `max_possible_fitness`, the maximum fitness achievable, in case a certain goal exists and the user would like the algorithm to stop after reaching it, defaults to `genes_per_individual` when unspecified.
- `survivors_percentage`, the percentage of individuals that survive any `guesser.__select()` method call, defaults to 25 when unspecified.
- `mutation_chance_percentage`, the probability percentage of any gene to be re generated at random during the `guesser.__reproduce()` method call, defaults to 1.0 when unspecified.
- `seed`, the random generator's seed, defaults to `None` when unspecified.

.

The method `guesser.generational_step()` executes a single iteration of the algorithm, running the methods `guesser.__select()`, `guesser.__reproduce()`, and lastly `guesser.__evaluate()`, in that order.

Selection implies keeping a percentage of the population by sequentially deleting the individuals with the lowest scores until that is achieved.

Reproduction means re-using the information of the survivors to generate a new population, where every individual is born from a combination of combining the parent's genes and random mutations controlled by a probability.

Evaluation implies re-calling the evaluation function on all individuals, updating their fitness statuses for the next iteration.

This process repeats until the user evaluates the resulting best individual, accessible through the `guesser.get_best_individual()` method or indirectly thorough the `guesser.get_max_fitness()` method, as good enough or until the algorithm recognizes that the maximum possible fitness has been found.

For making the recognition of this event easy, `guesser.generational_step()` returns `True` after the maximum possible fitness has been found and `False` otherwise.

So a loop can take the form:

```
1 i = 1
2 while guesser.generational_step():
3     print("The current best fitness in the population is = " + str(guesser.
4         get_max_fitness()))
5     i += 1
6 print("The maximum fitness was found after " + str(i) + " iterations")
```

2.3 Implementation

2.3.1 The Genetic Guesser

Inside the `src/genetic_algorithms` package, in the `genetic_algorithm.py` file a Python class was implemented:

```
1 class GeneticGuesser:
2
3     def __init__(self, individuals_amount: int,
4                   genes_per_individual: int,
5                   gene_alphabet: list,
6                   evaluation_function=None,
7                   max_possible_fitness: int = None,
8                   survivors_percentage: float = 25,
9                   mutation_chance_percentage: float = 1,
10                  seed: int = None):...
11
12     @staticmethod
13     def builder():...
14
15     def change_evaluation_function(self, new_function) -> None:...
16
17     def change_max_possible_fitness(self, fitness: int) -> None:...
18
19     @staticmethod
20     def _null_evaluation_function(word: list) -> int:...
21
22     def __evaluate(self) -> None:...
23
24     def is_done(self) -> bool:...
25
26     def get_max_fitness(self) -> int:...
27
28     def get_best_individual(self) -> list:...
29
30     def __select(self) -> None:...
31
32     def __reproduce(self) -> None:...
33
34     def generational_step(self) -> bool:...
35
```

```
36     class Builder:
37
38         def __init__(self):...
39
40         def with_individuals(self, amount: int):...
41
42         def with_genes_amount(self, amount_per_individual: int):...
43
44         def with_alphabet(self, alphabet: list):...
45
46         def with_evaluation_function(self, new_function):...
47
48         def with_max_fitness(self, fitness: int):...
49
50         def with_survivors(self, percentage: float):...
51
52         def with_mutation_chance(self, percentage: float):...
53
54         def with_seed(self, seed: int):...
55
56         def build(self):...
```

This code is fully documented within it's package, so only some small examples of usage are included here

An example of a construction for this class is:

```
1 guesser = GeneticGuesser.Builder()\
2     .with_alphabet(alphabet)\
3     .with_individuals(individual_length*3)\
4     .with_genes_amount(individual_length)\
5     .with_evaluation_function(lambda gene: individual_evaluation(gene,\
6         base_individual))\
7     .with_survivors(25.0)\
8     .with_mutation_chance(5.0)\
9     .with_seed(seed)\
10    .build()
```

Which creates a `guesser` with the specified attributes, where `with_alphabet`, `with_individuals` and `with_genes_amount` are obligatory calls, since those are obligatory parameters for the `guesser`'s native constructor, or else `Builder.build()` raises an `AttributeError`

An example of a learning loop for the `guesser` is then the same as it was decided in the solution scheme:

```
1 i = 1
2 while guesser.generational_step():
```

```
3     print("The current best fitness in the population is = " + str(guesser.  
        get_max_fitness()))  
4     i += 1  
5 print("The maximum fitness was found after " + str(i) + " iterations")
```

2.3.2 Word Guessing Demonstration

When executing the `src/genetic_algorithms/genetic_algorithm.py` file, it's main function will run some plotted demonstrations of the genetic algorithm.

This demonstrations are different variations of the Word Guessing problem.

The `main` function receives as a parameter the alphabet from which the words will be built, along with parameters for the guesser and some configurations for producing and saving plots.

Then it generates a random word and lets the genetic guesser iterate until it's produced it.

For this an evaluation function called `word_comparator(word_1, word_2)` exists, which compares individuals in the guesser's population to the target word, and is passed to the guesser upon construction.

`main` returns the total amount of iterations that were needed for guessing the word.

Some example code of one of these tests is:

```
1     # ----- Finding Length -----  
2     ""  
3     Test for comparing the amount of iterations needed when word guessing for  
        different length of a binary word  
4     ""  
5  
6     print("\nRunning length test")  
7  
8     dm.clear_dir(length_test_plots_saving_directory)  
9  
10    diffs = []  
11    iterations = []  
12  
13    for diff in range(points):  
14        p = (float(diff) + 1) / points  
15        diffs.append(100 + int(100*p))  
16        iterations.append(main(diffs[diff], binary_alphabet, 3*diffs[diff], 25.0,  
                                5.0, seed=s, save_directory=length_test_plots_saving_directory))  
17        s += 1  
18
```

```

19 plot_result(diffs, iterations, "Length vs Iterations needed", "Length", "
    Iterations", save_directory=length_test_plots_saving_directory)

```

Where `dm.clear_dir(...)` is a utility function for clearing directories, `main(...)` runs the genetic algorithm and `plot_result(...)` generates a plot of the amount of iterations for each word length.

2.3.3 The N-Queen Problem

The code for solving the N-Queen Problem can be found in `src/genetic_algorithms/n_queen_optimizer.py`.

First, the model for an individual in the guesser's population is a `list` array of coordinates for the queen pieces, and therefore with the length of twice the amount of pieces.

The fitness function for the algorithm is as follows:

```

1 def fitness_evaluation(queens_configuration: list) -> int:
2     """
3     Simple function for evaluating the fitness of a list of queens
4
5     The analogy for this calculation is as follows:
6
7     Imagine the pieces as nodes of a undirected graph, where only the pieces that
8     can attack each other are connected
9     by edges, the most edges that the graph can have is the sum of every positive
10    integer from 1 to the amount of queens
11
12    This case scenario could happen if the queen pieces are all in the same column
13    , for example
14
15    The function returns that number, the sum of all positive integers from 1 to
16    the amount of queens, minus the amount
17    of actual edges in the graph for the parameter queens_configuration
18
19    Therefore the more actual edges found, the less fitness
20
21    So if all queens were found on the same column, or row, or a diagonal, the
22    function returns 0
23
24    :param queens_configuration: A list of length 2*<the amount of queens>, with
25    their coordinates
26    :return: The fitness of the configuration
27    """

```

```
23     # Ideal value
24     fitness = sum(range(len(queens_configuration)))
25     for i in range(int(len(queens_configuration) / 2)):
26         x = queens_configuration[2 * i]
27         y = queens_configuration[2 * i + 1]
28         for j in range(int((len(queens_configuration)/2)-i-1)):
29             j += i+1
30             x2 = queens_configuration[2 * j]
31             y2 = queens_configuration[2 * j + 1]
32             # Edge found
33             if x == x2 or y == y2 or abs(x-x2) == abs(y-y2):
34                 fitness -= 1
35     return fitness
```

The file's main function runs when the file is executed, constructing a **GeneticGuesser** object with the aforementioned function and the maximum fitness that it allows, and letting it loop until a solution is found.

The loop resets after a certain amount of iterations, to ensure the algorithm doesn't stay too long iterating over some local optima.

Finally a plot for the configuration found is generated and saved.

Again, more information can be found in the file's documentation.

3 Results

3.1 Word Guessing Demonstration

[1](#)

3.2 The N-Queen Problem

4 Discussion

5 Conclusions

6 Figures

6.1 Word Guessing

6.1.1 Word length test

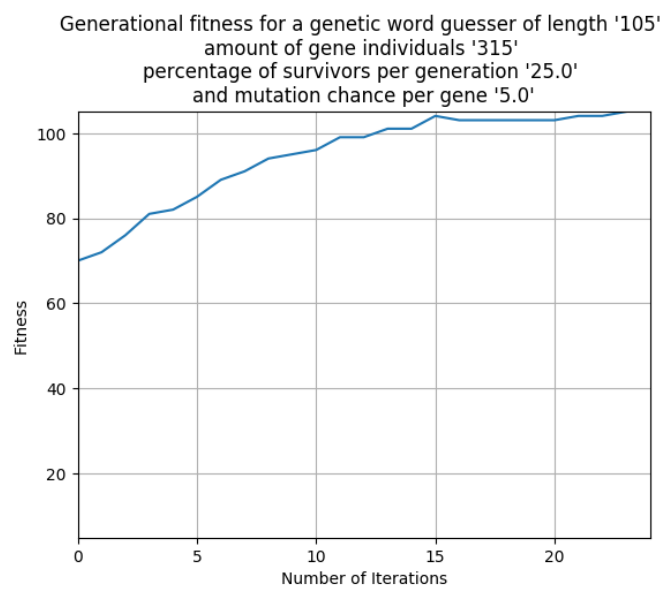


Figure 1: Test using binary alphabet for Length 105

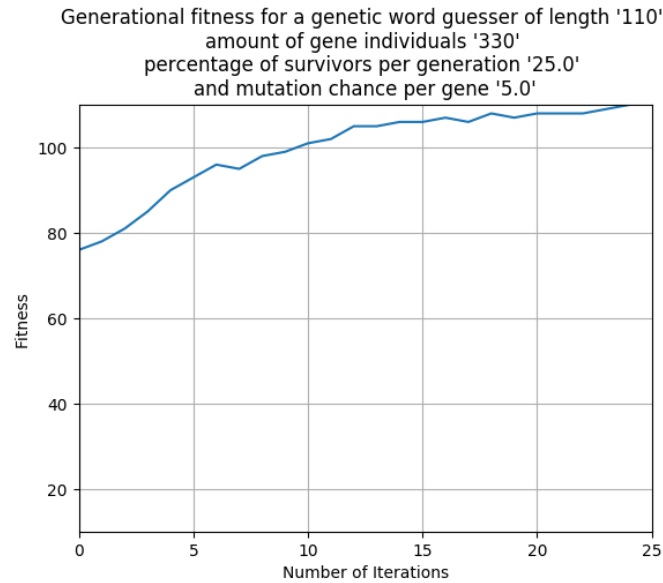


Figure 2: Test using binary alphabet for Length 110

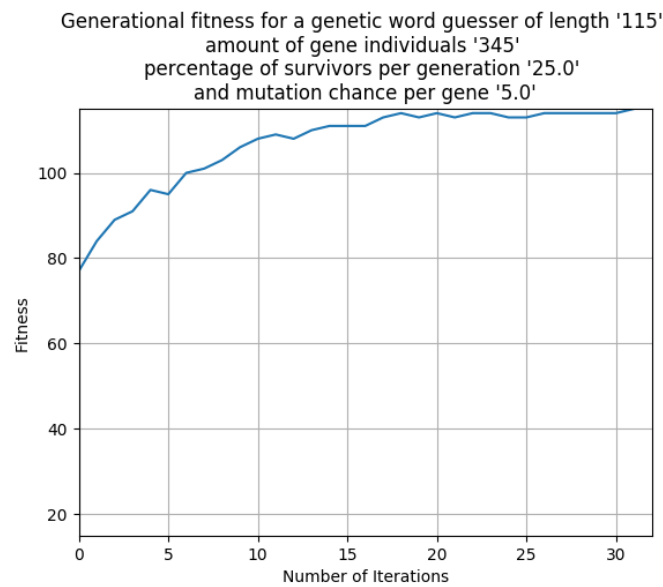


Figure 3: Test using binary alphabet for Length 115

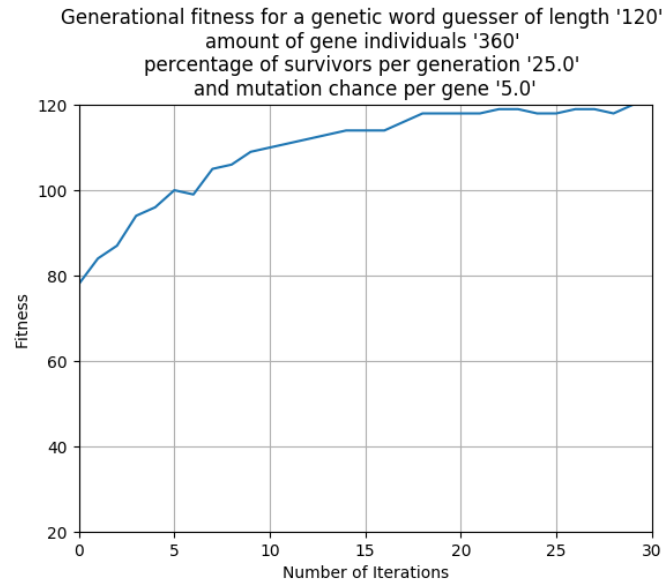


Figure 4: Test using binary alphabet for Length 120

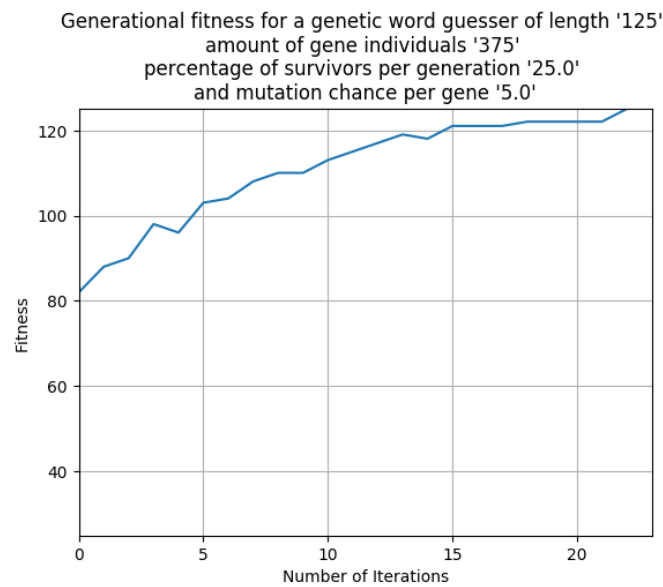


Figure 5: Test using binary alphabet for Length 125

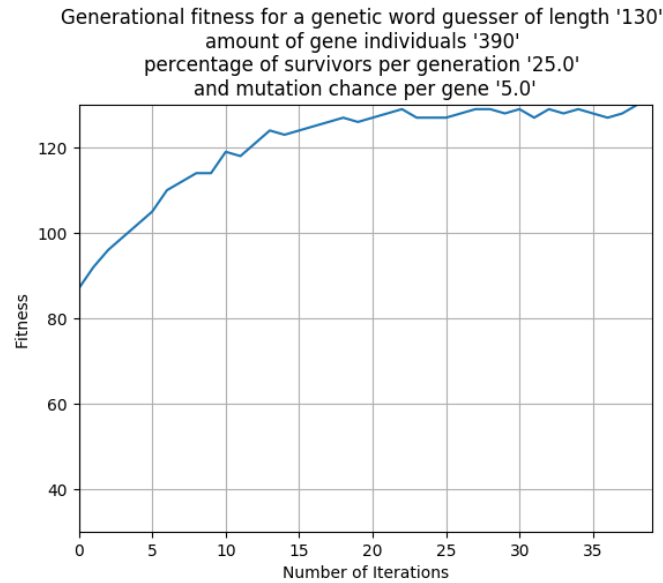


Figure 6: Test using binary alphabet for Length 130

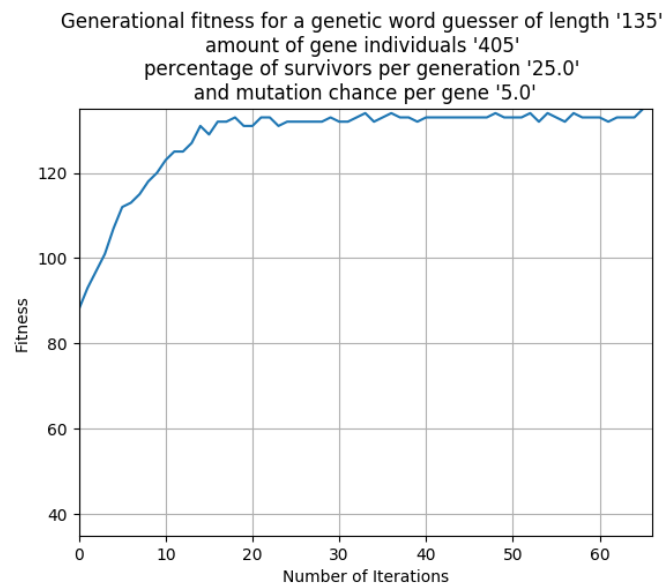


Figure 7: Test using binary alphabet for Length 135

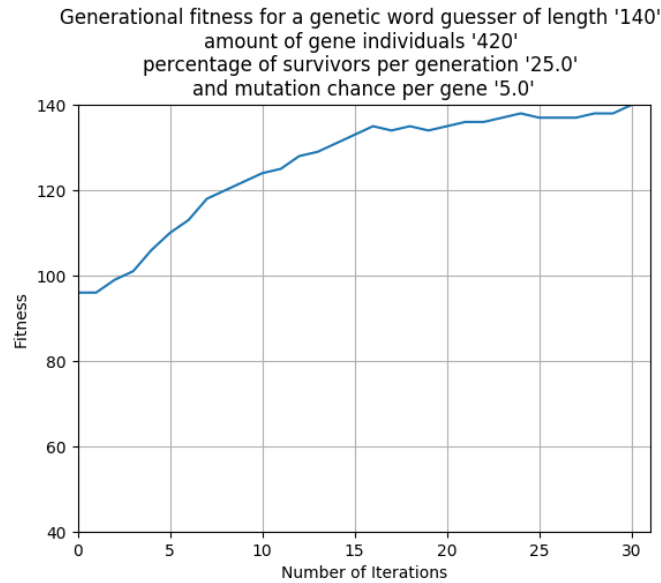


Figure 8: Test using binary alphabet for Length 140

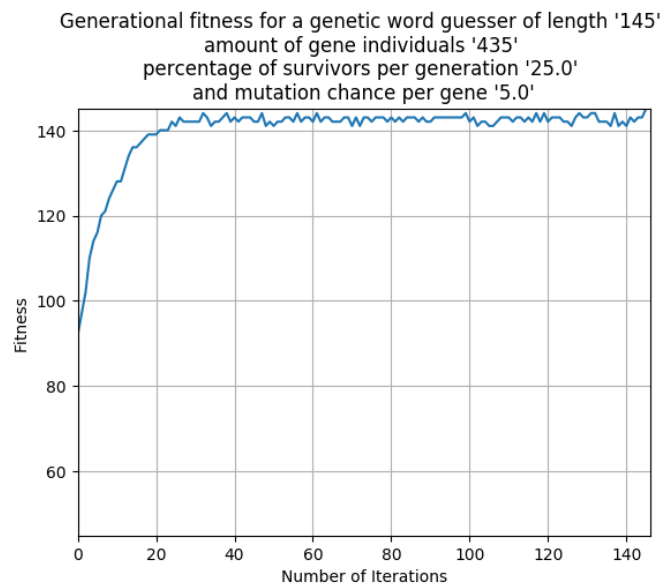


Figure 9: Test using binary alphabet for Length 145

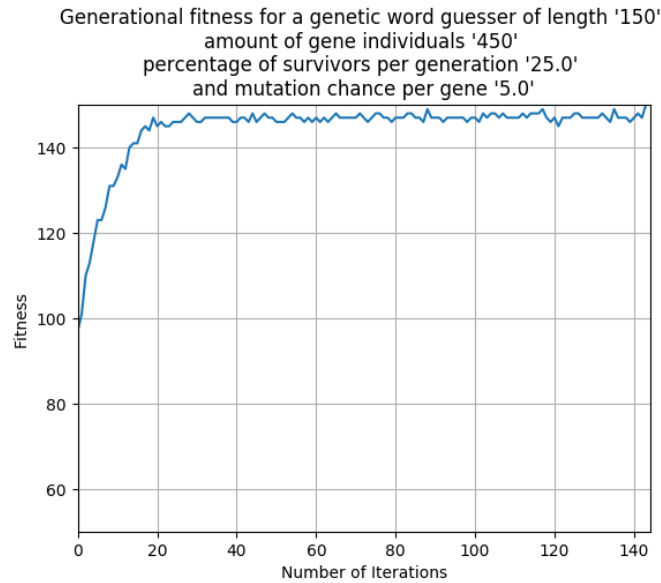


Figure 10: Test using binary alphabet for Length 150

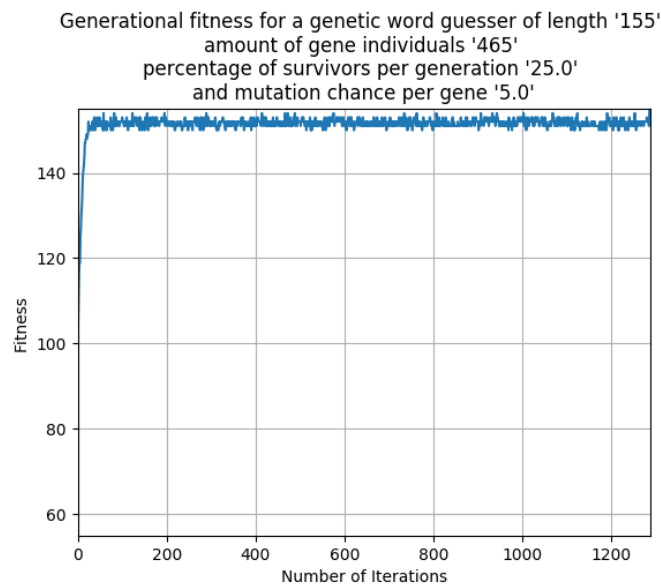


Figure 11: Test using binary alphabet for Length 155

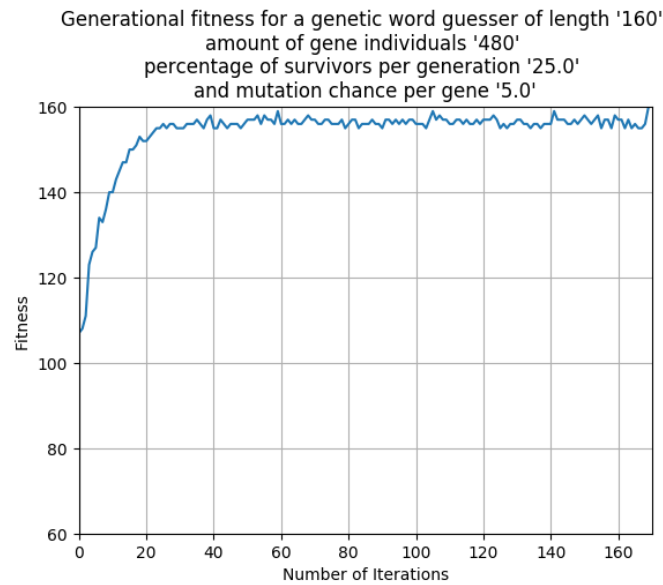


Figure 12: Test using binary alphabet for Length 160

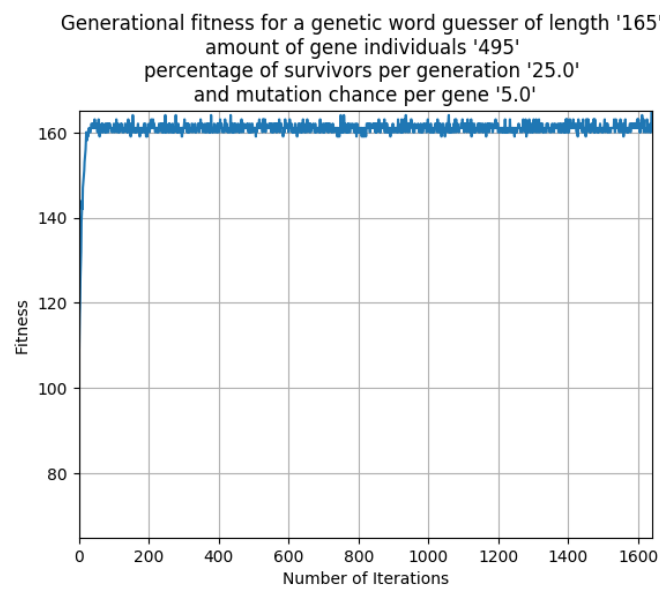


Figure 13: Test using binary alphabet for Length 165

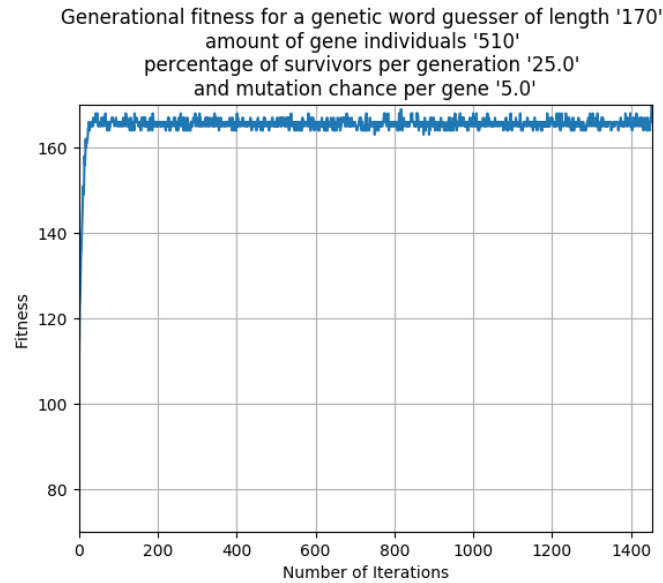


Figure 14: Test using binary alphabet for Length 170

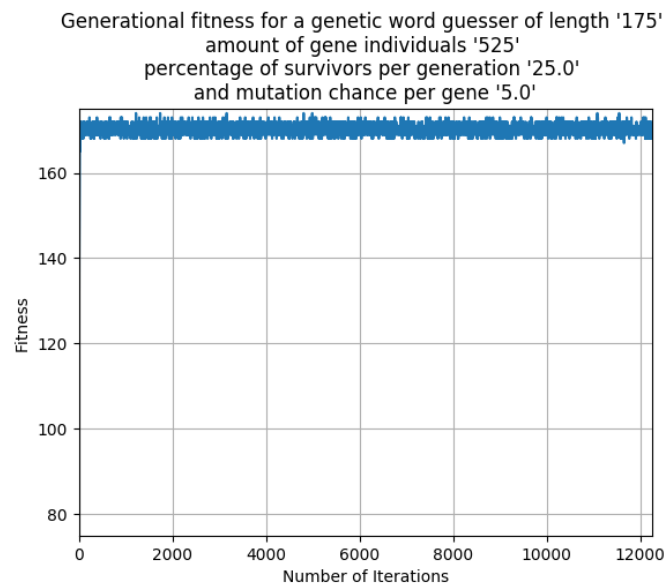


Figure 15: Test using binary alphabet for Length 175

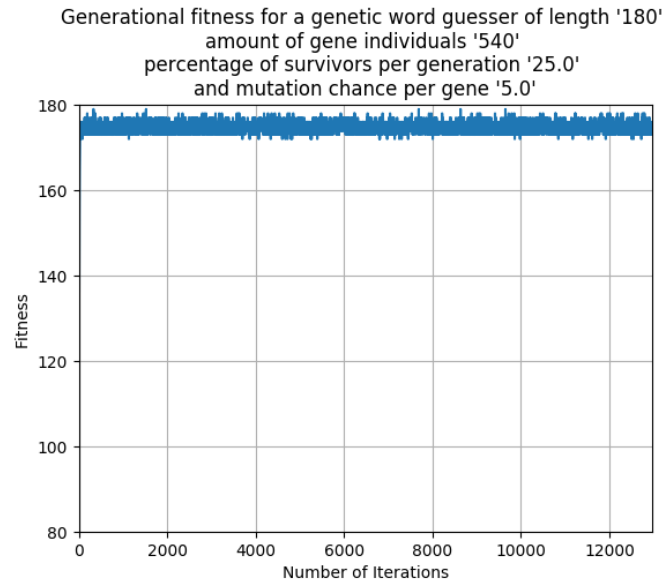


Figure 16: Test using binary alphabet for Length 180

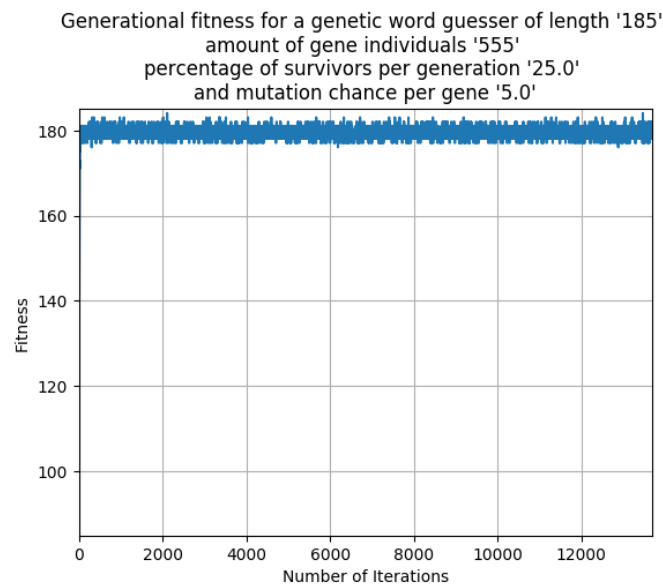


Figure 17: Test using binary alphabet for Length 185

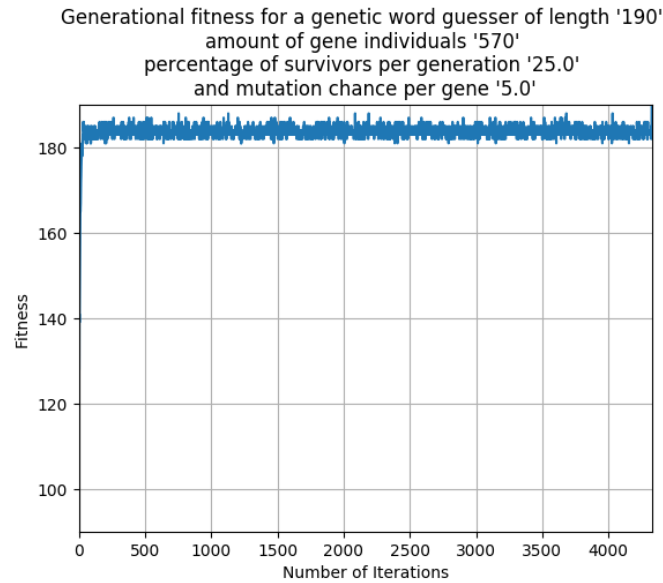


Figure 18: Test using binary alphabet for Length 190

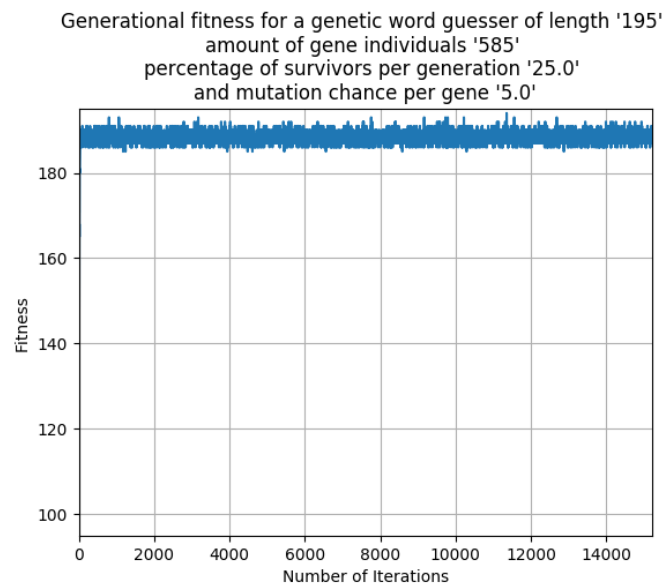


Figure 19: Test using binary alphabet for Length 195

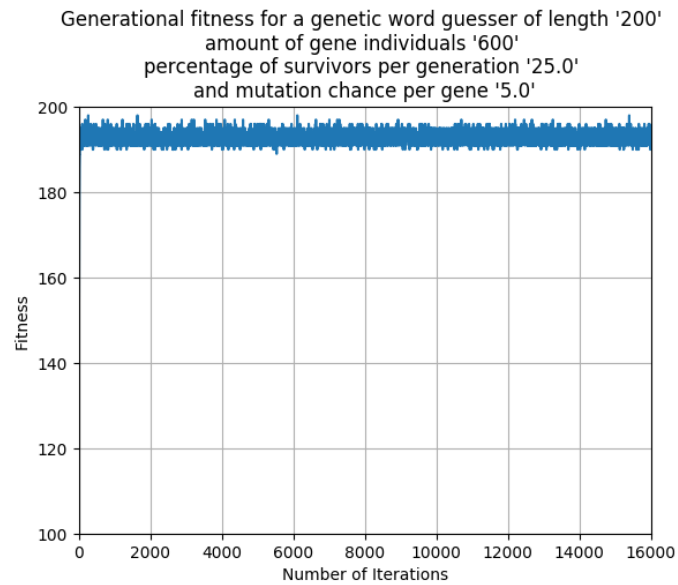


Figure 20: Test using binary alphabet for Length 200

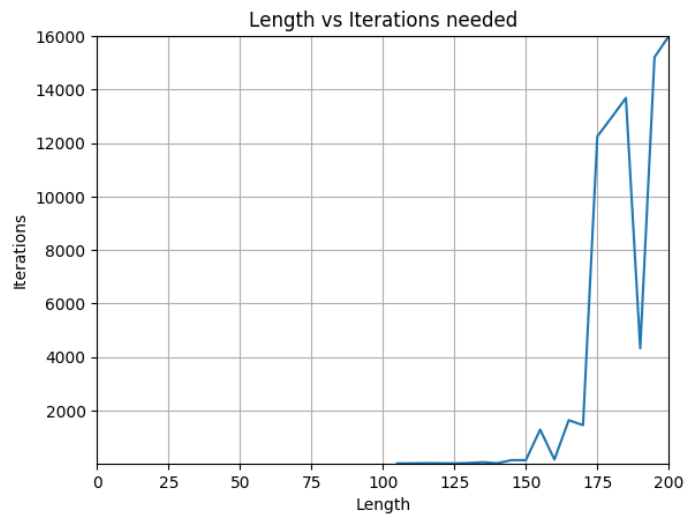


Figure 21: All length tests for binary alphabet

6.1.2 Population size test

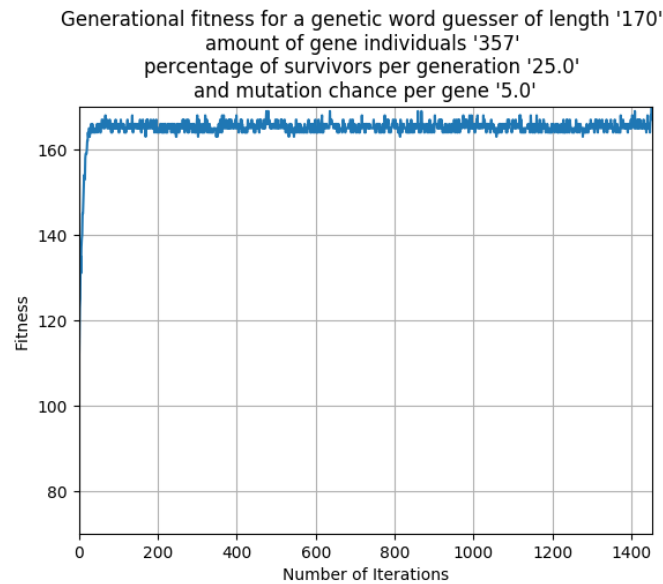


Figure 22: Test using binary alphabet for Population 357

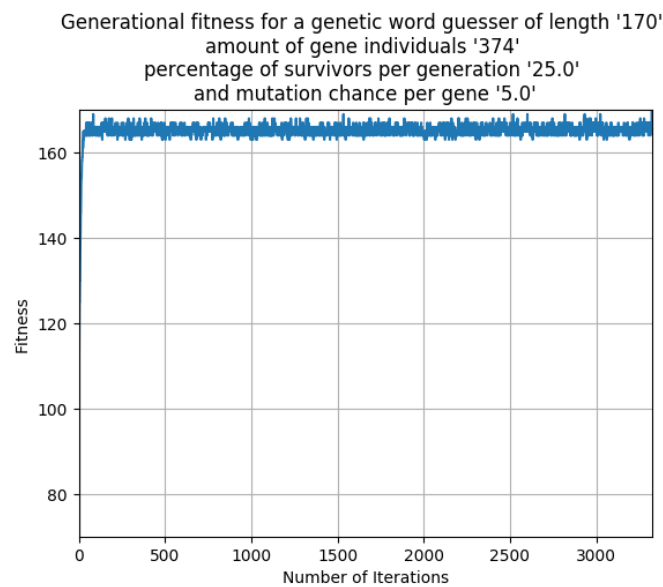


Figure 23: Test using binary alphabet for Population 374

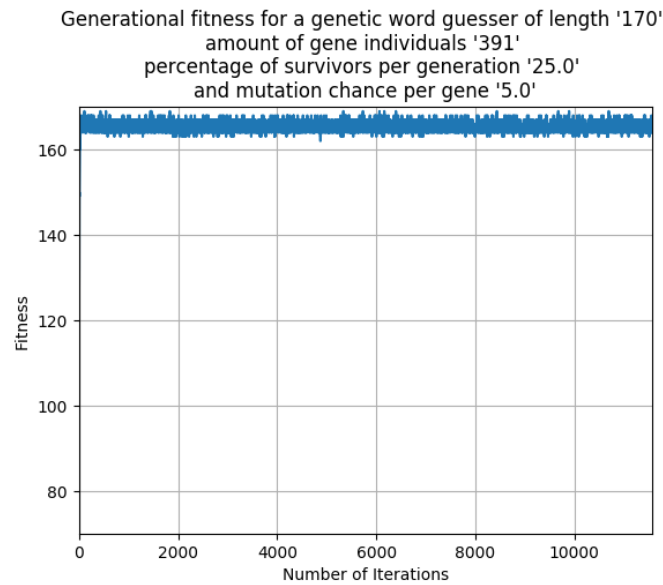


Figure 24: Test using binary alphabet for Population 391

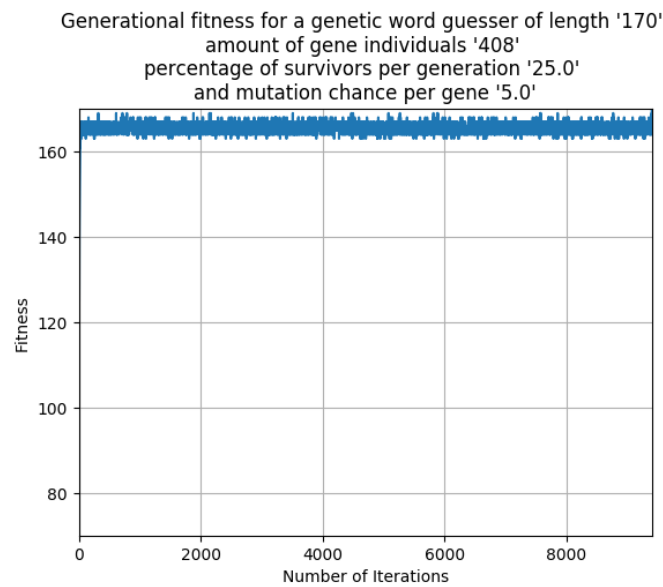


Figure 25: Test using binary alphabet for Population 408

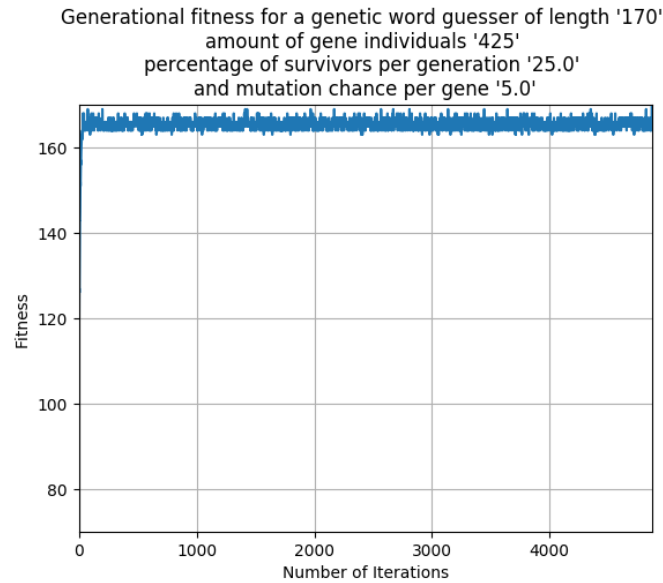


Figure 26: Test using binary alphabet for Population 425

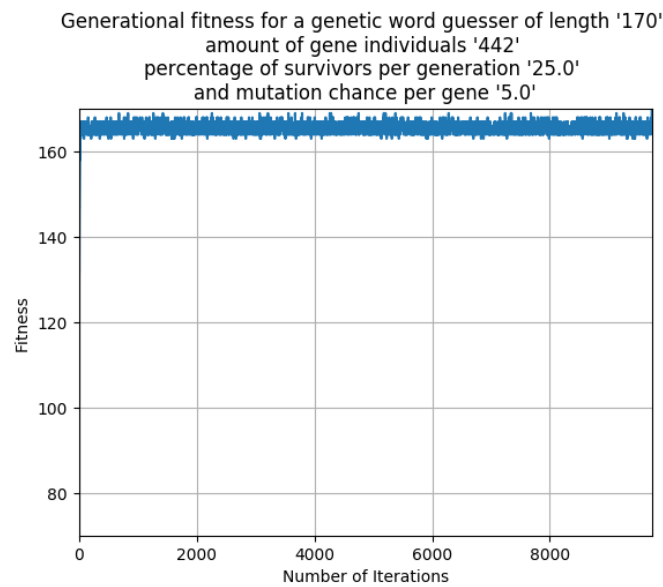


Figure 27: Test using binary alphabet for Population 442

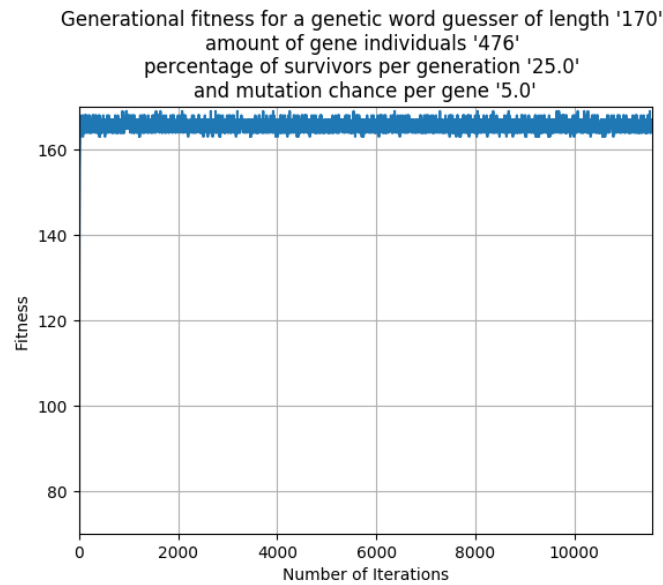


Figure 28: Test using binary alphabet for Population 476

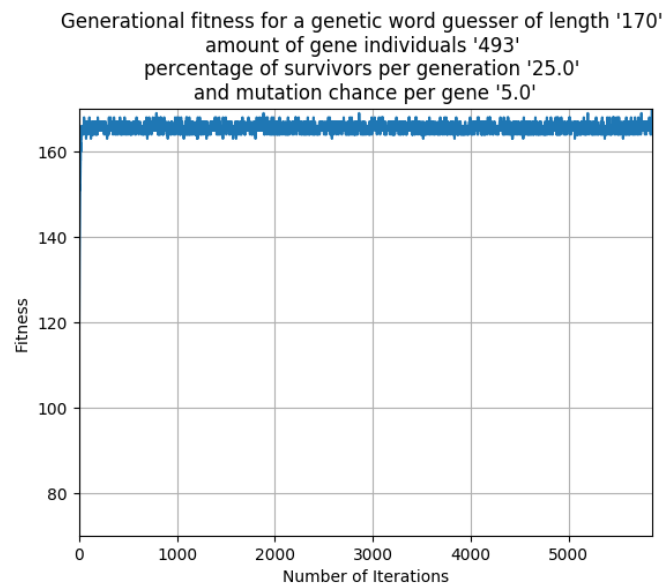


Figure 29: Test using binary alphabet for Population 493

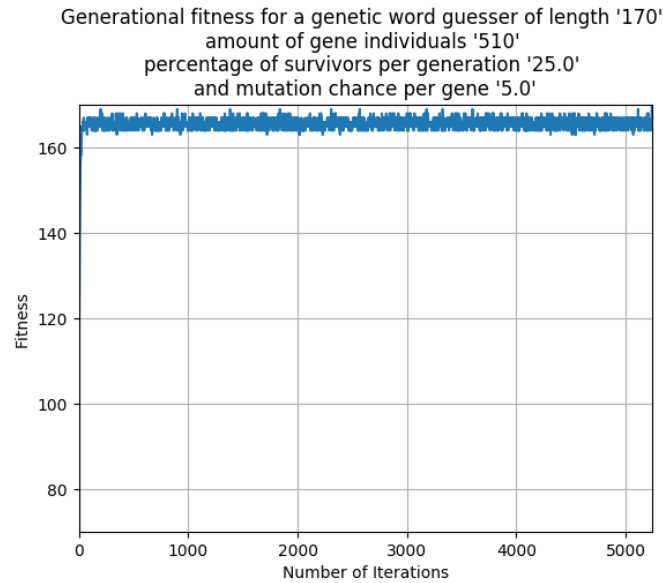


Figure 30: Test using binary alphabet for Population 510

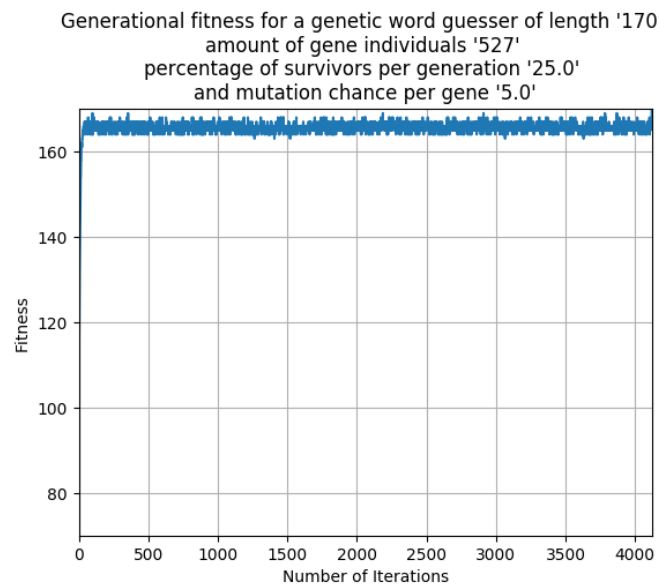


Figure 31: Test using binary alphabet for Population 527

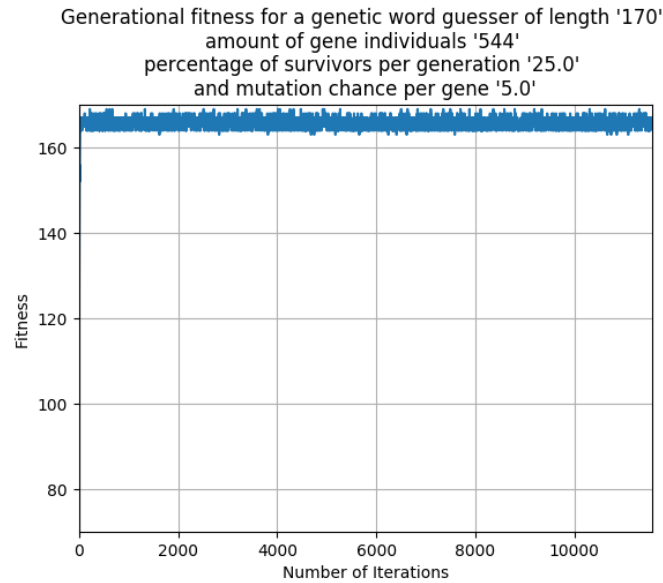


Figure 32: Test using binary alphabet for Population 544

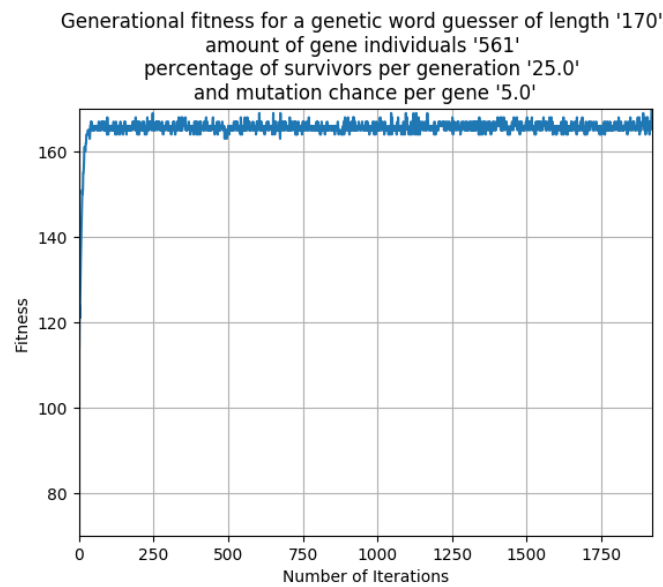


Figure 33: Test using binary alphabet for Population 561

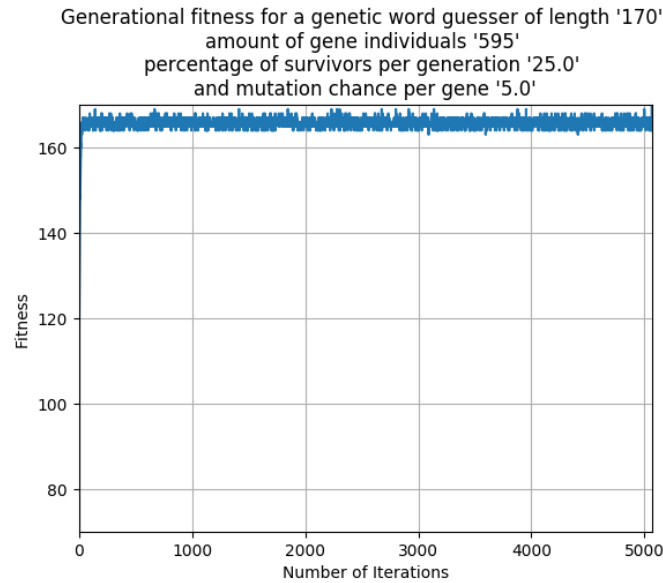


Figure 34: Test using binary alphabet for Population 595

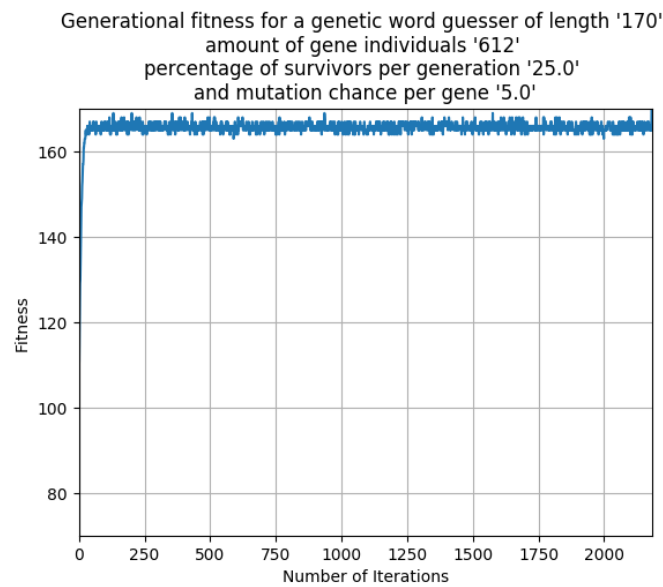


Figure 35: Test using binary alphabet for Population 612

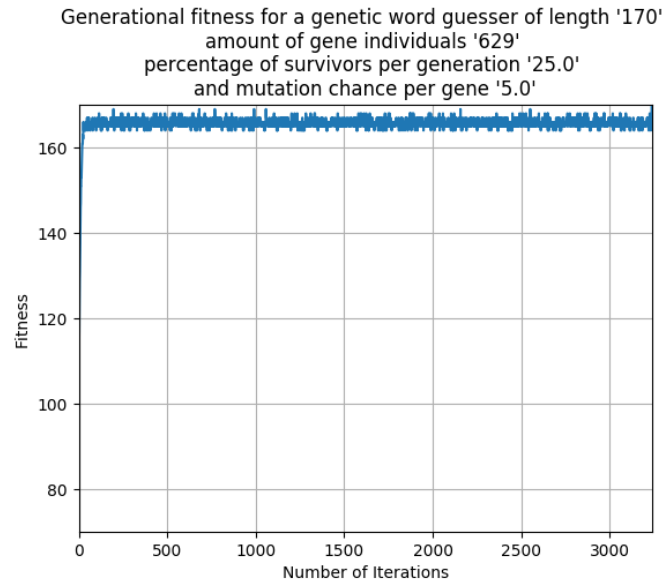


Figure 36: Test using binary alphabet for Population 629

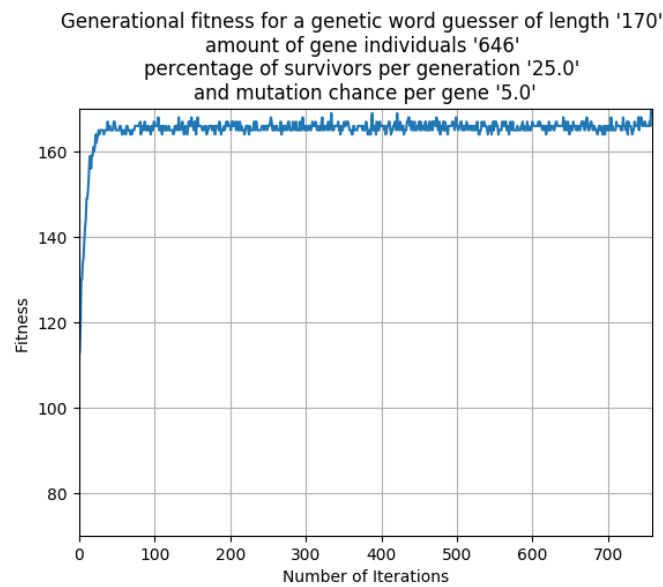


Figure 37: Test using binary alphabet for Population 646

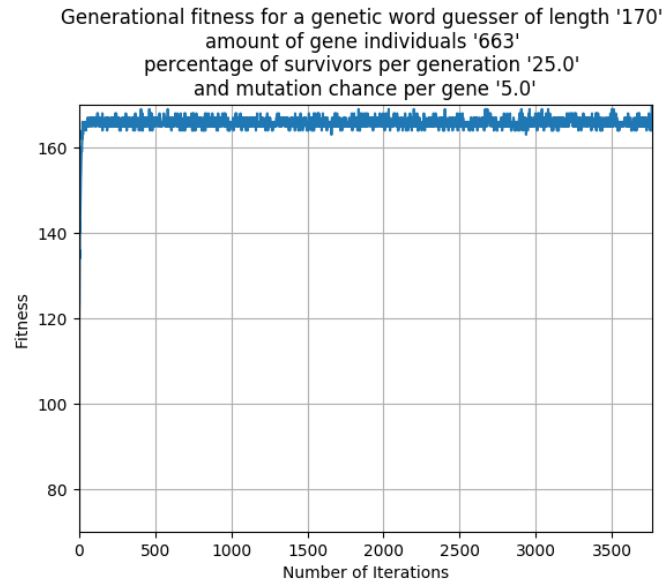


Figure 38: Test using binary alphabet for Population 663

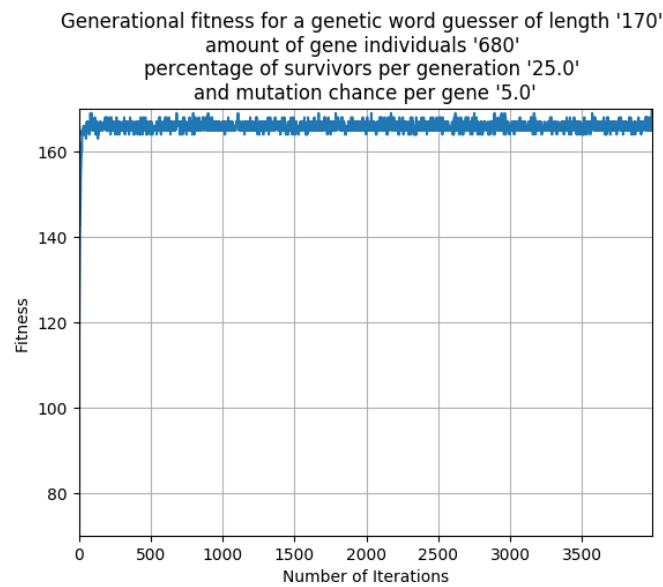


Figure 39: Test using binary alphabet for Population 680

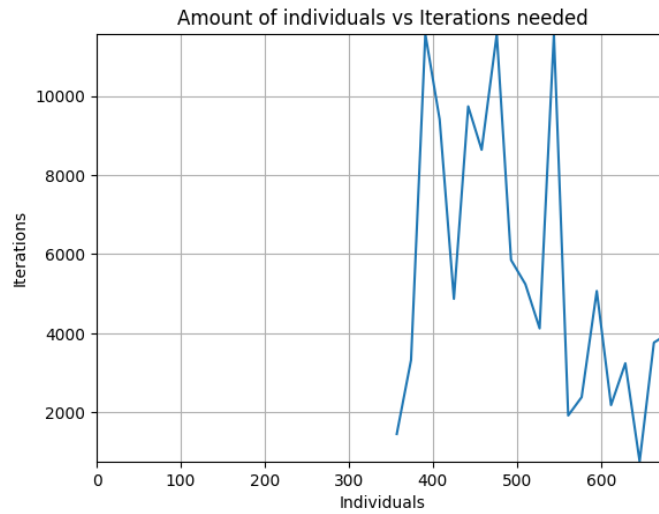


Figure 40: All population tests for binary alphabet

6.1.3 Survivors percentage test

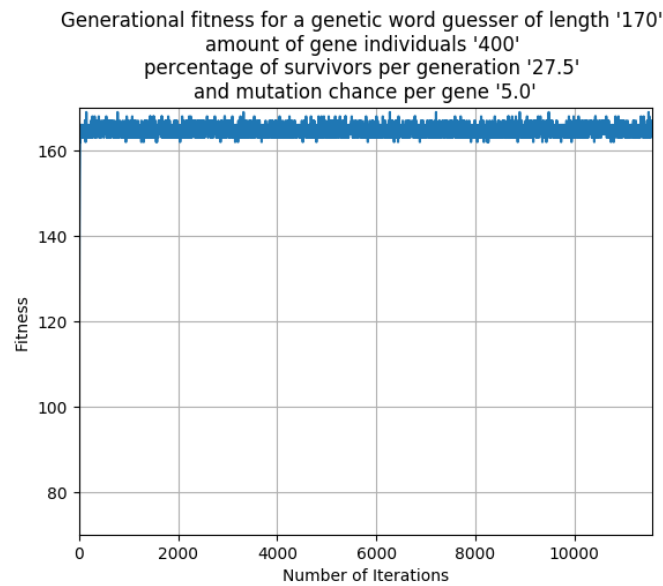


Figure 41: Test using binary alphabet for Survivors Percentage 27.5

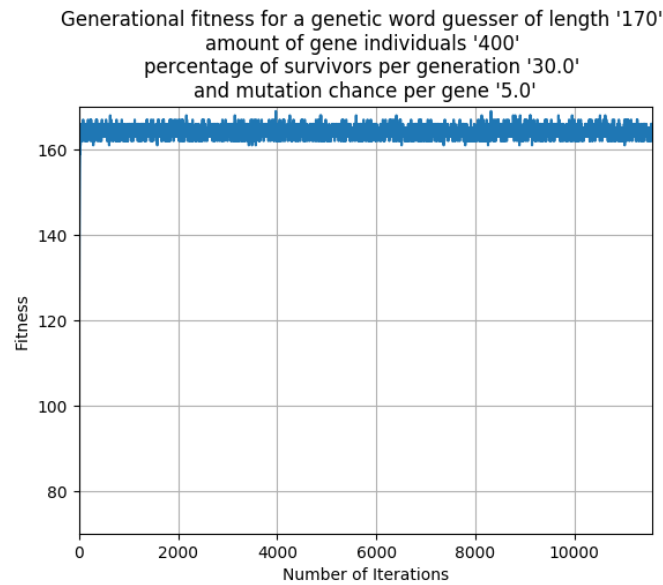


Figure 42: Test using binary alphabet for Survivors Percentage 30.0

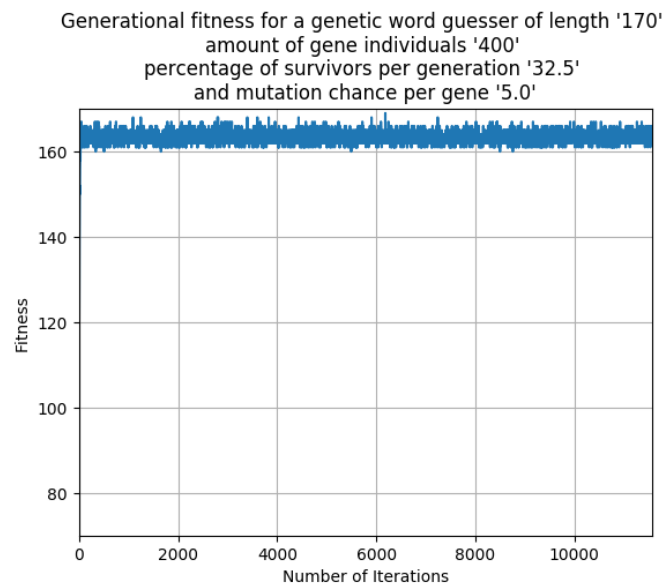


Figure 43: Test using binary alphabet for Survivors Percentage 32.5

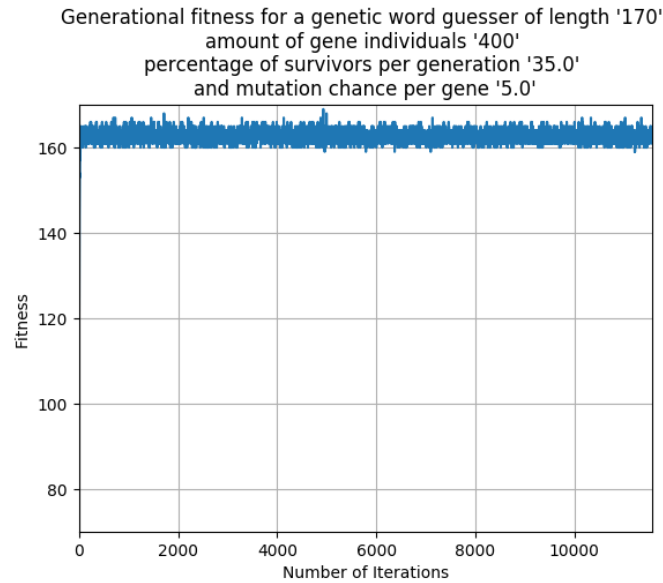


Figure 44: Test using binary alphabet for Survivors Percentage 35.0

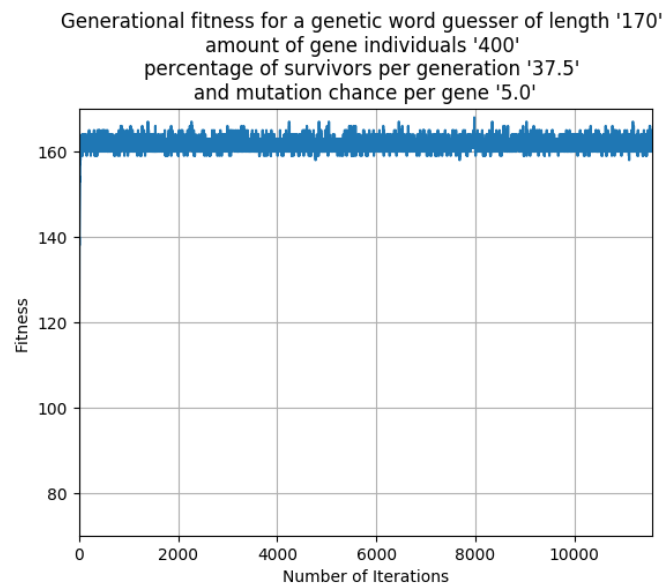


Figure 45: Test using binary alphabet for Survivors Percentage 37.5

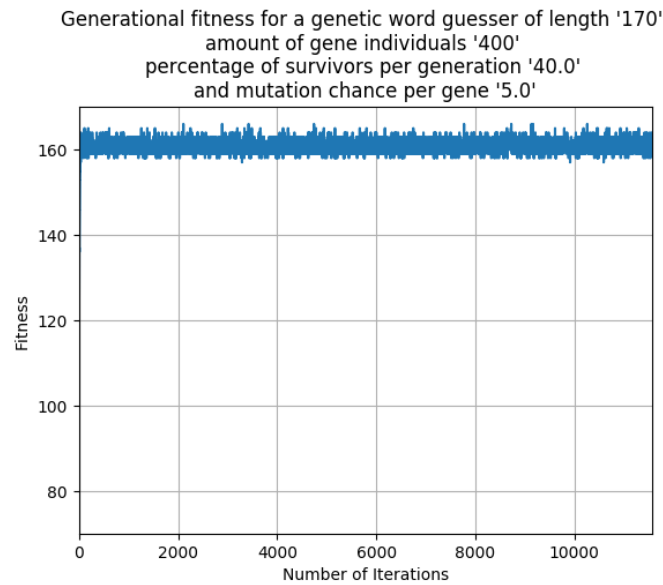


Figure 46: Test using binary alphabet for Survivors Percentage 40.0

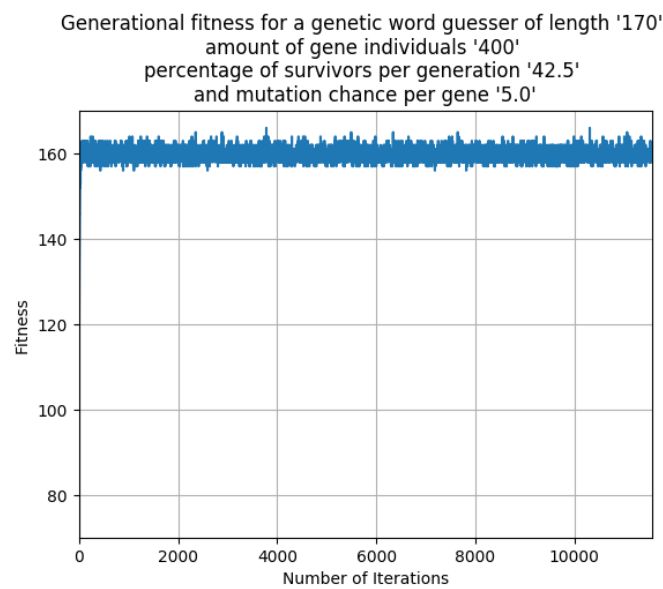


Figure 47: Test using binary alphabet for Survivors Percentage 42.5

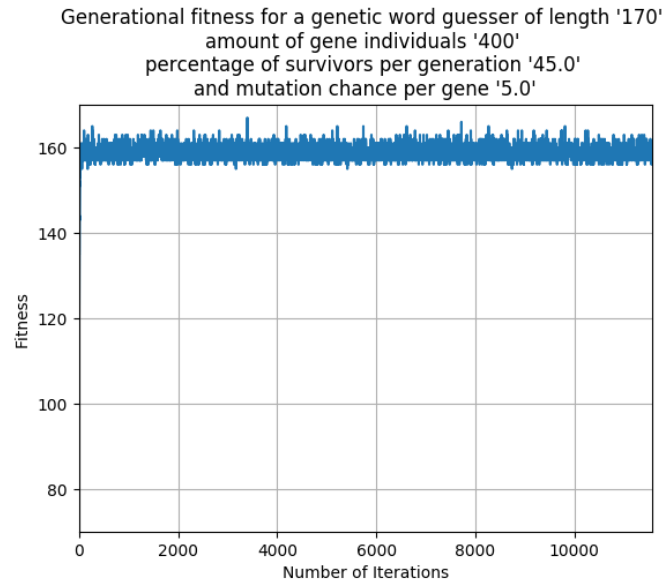


Figure 48: Test using binary alphabet for Survivors Percentage 45.0

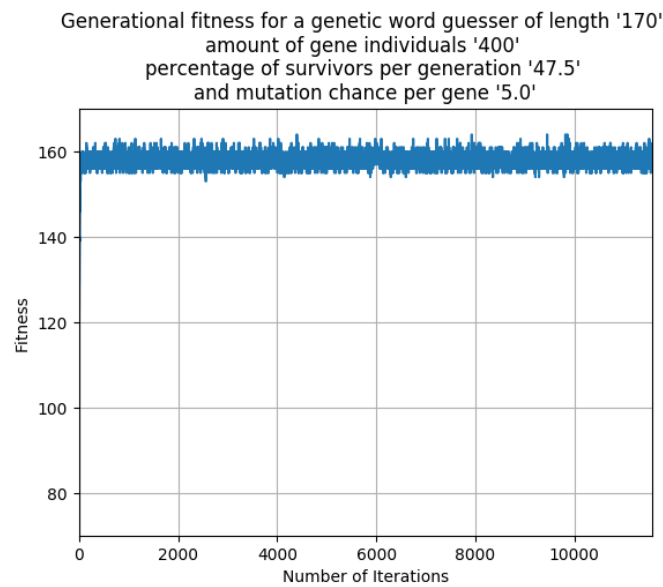


Figure 49: Test using binary alphabet for Survivors Percentage 47.5

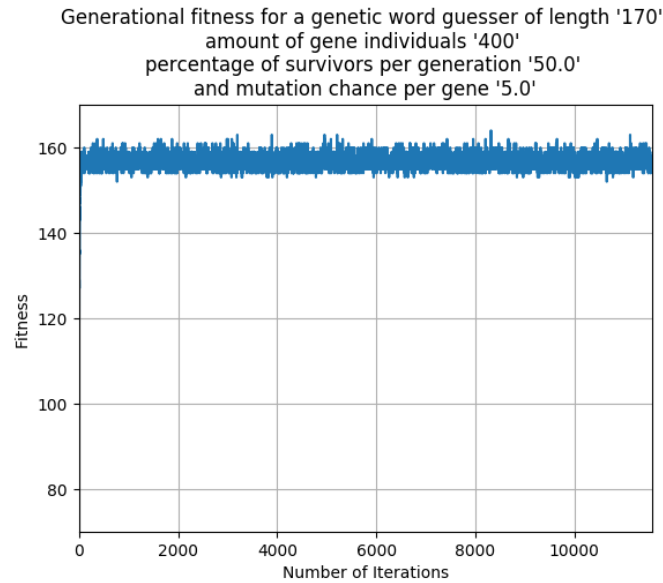


Figure 50: Test using binary alphabet for Survivors Percentage 50.0

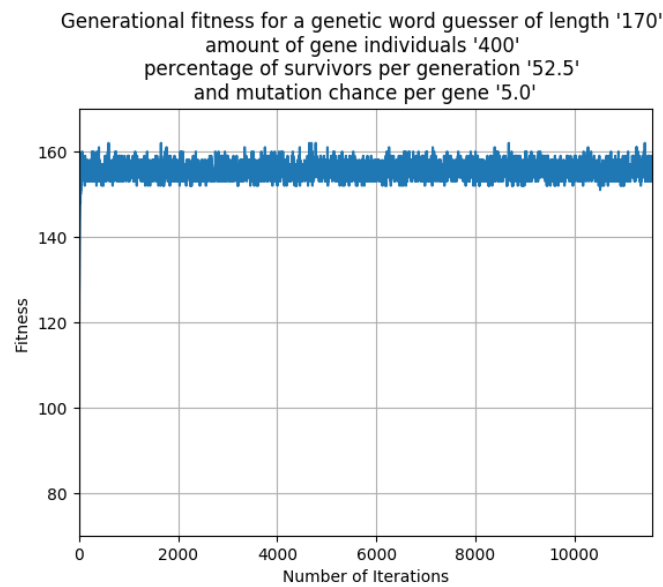


Figure 51: Test using binary alphabet for Survivors Percentage 52.5

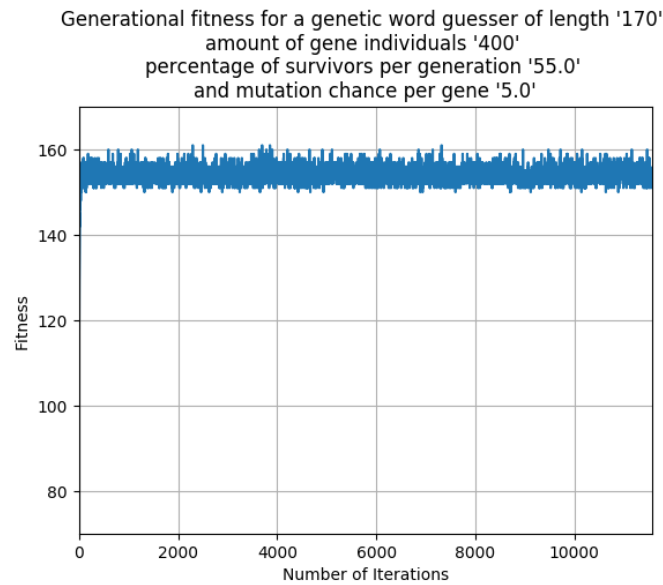


Figure 52: Test using binary alphabet for Survivors Percentage 55.0

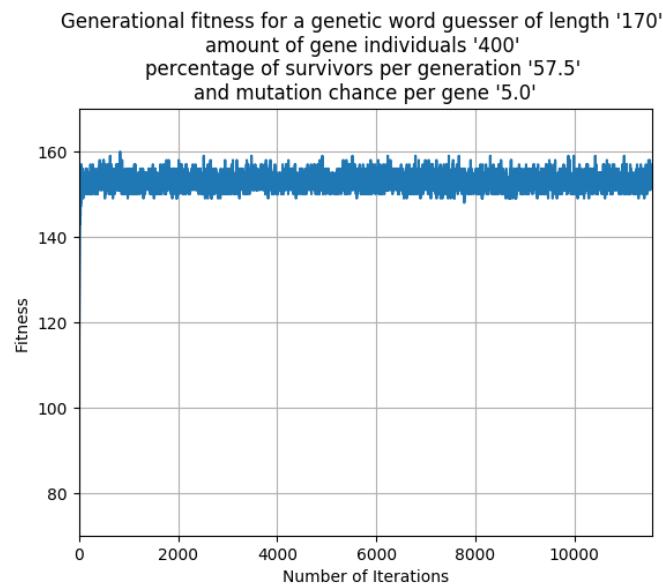


Figure 53: Test using binary alphabet for Survivors Percentage 57.5

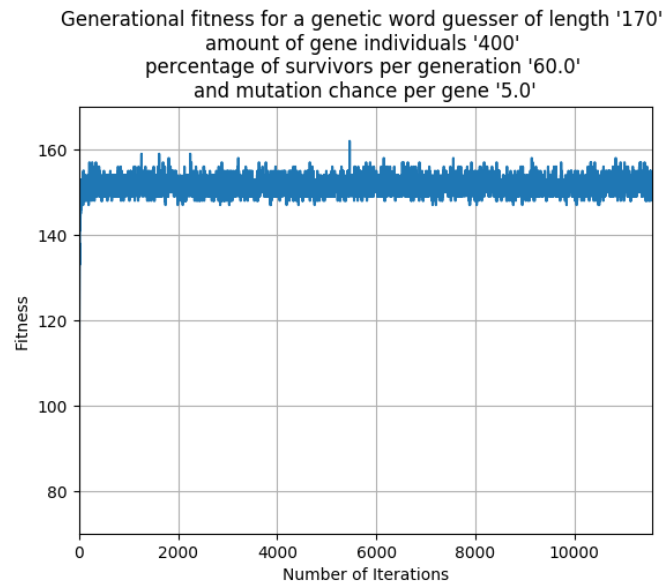


Figure 54: Test using binary alphabet for Survivors Percentage 60.0

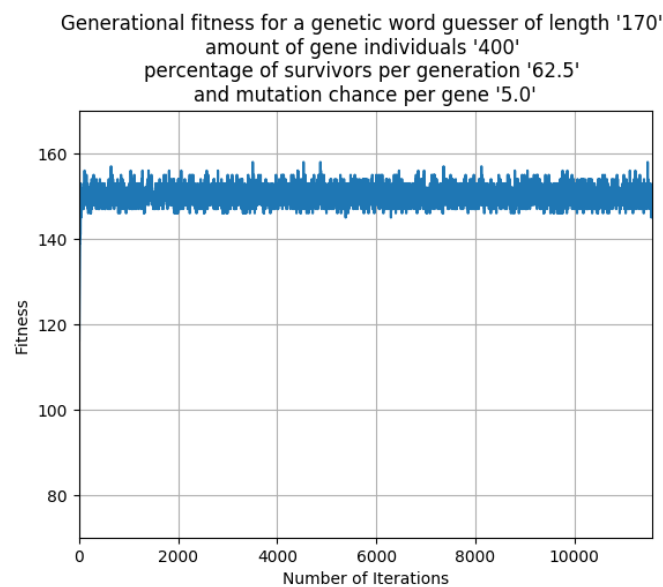


Figure 55: Test using binary alphabet for Survivors Percentage 62.5

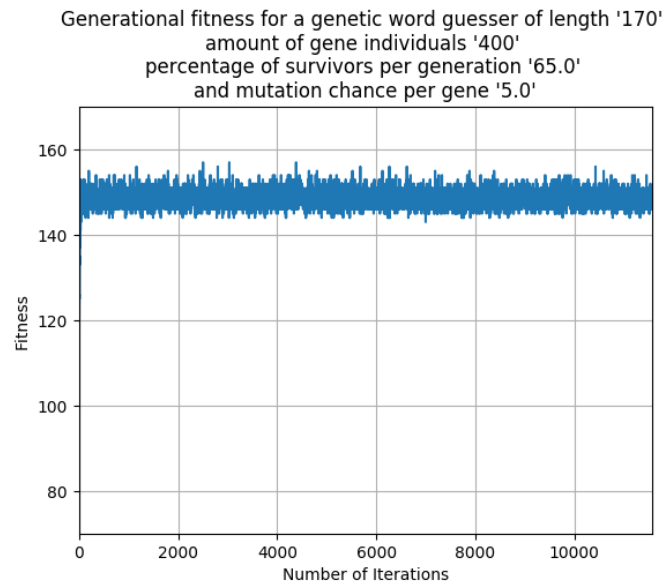


Figure 56: Test using binary alphabet for Survivors Percentage 65.0

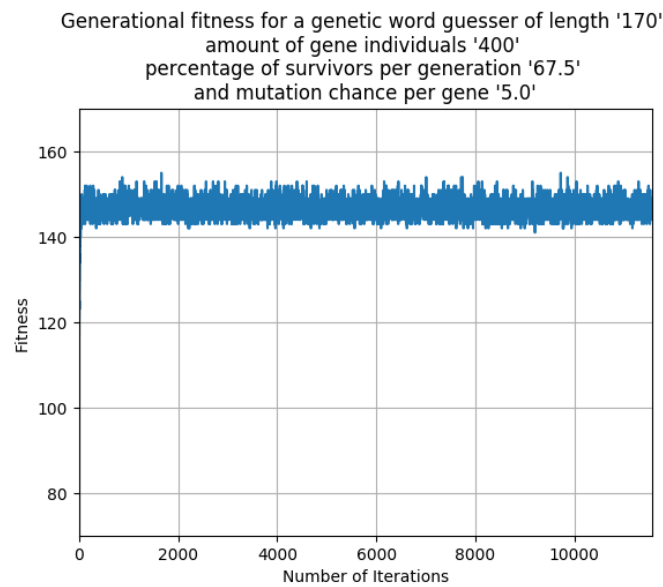


Figure 57: Test using binary alphabet for Survivors Percentage 67.5

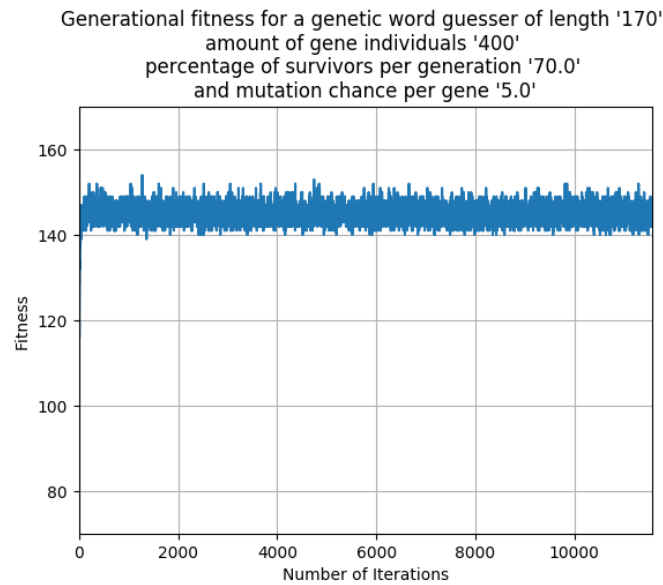


Figure 58: Test using binary alphabet for Survivors Percentage 70.0

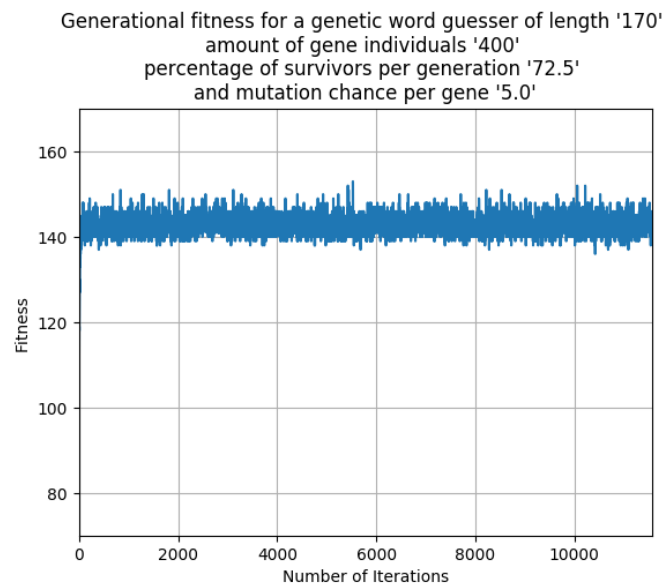


Figure 59: Test using binary alphabet for Survivors Percentage 72.5

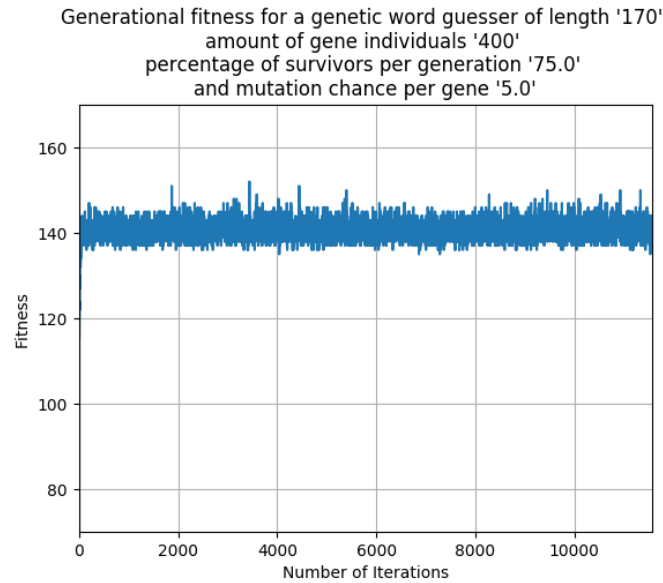


Figure 60: Test using binary alphabet for Survivors Percentage 75.0

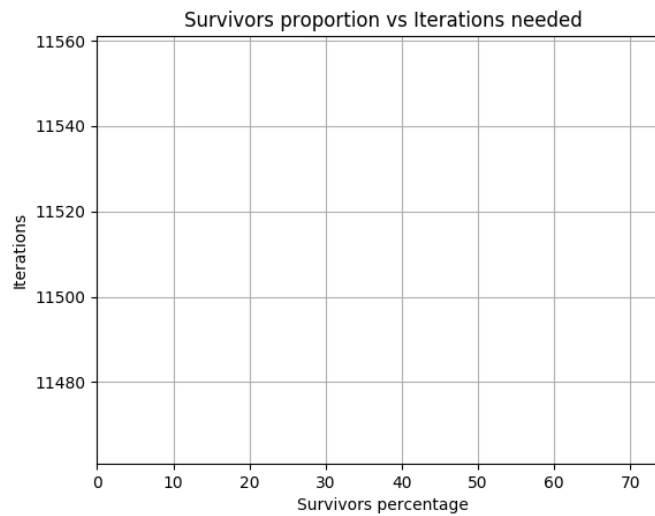


Figure 61: All Survivors Percentage tests for binary alphabet

6.1.4 Mutation chance test

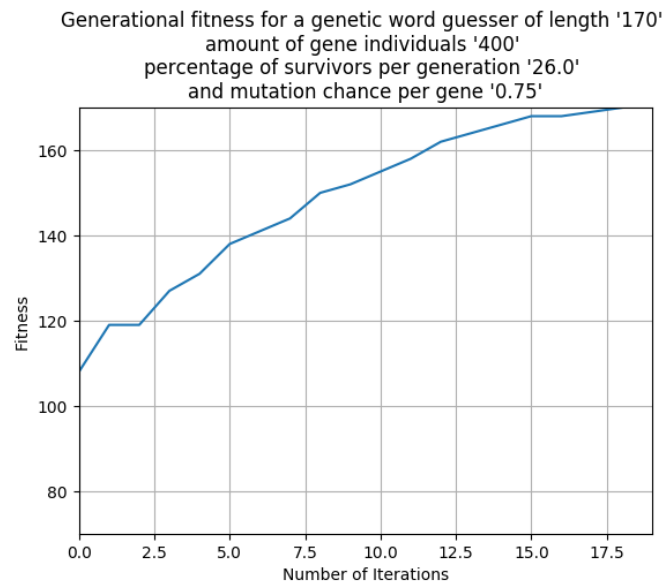


Figure 62: Test using binary alphabet for Mutation Chance 0

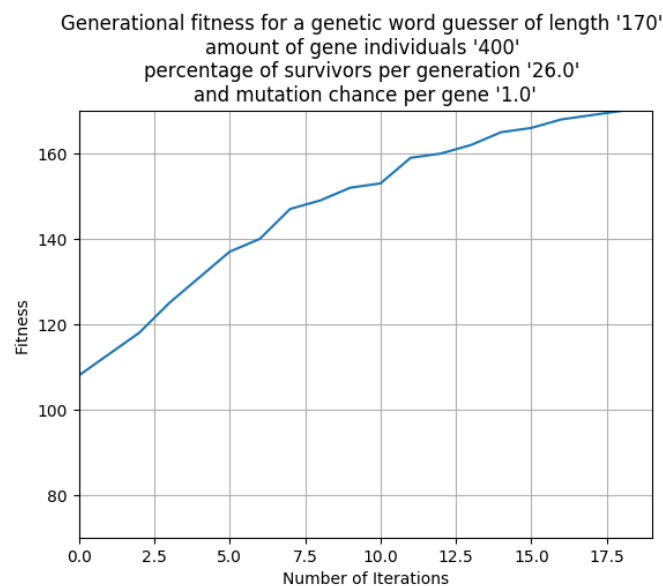


Figure 63: Test using binary alphabet for Mutation Chance 1

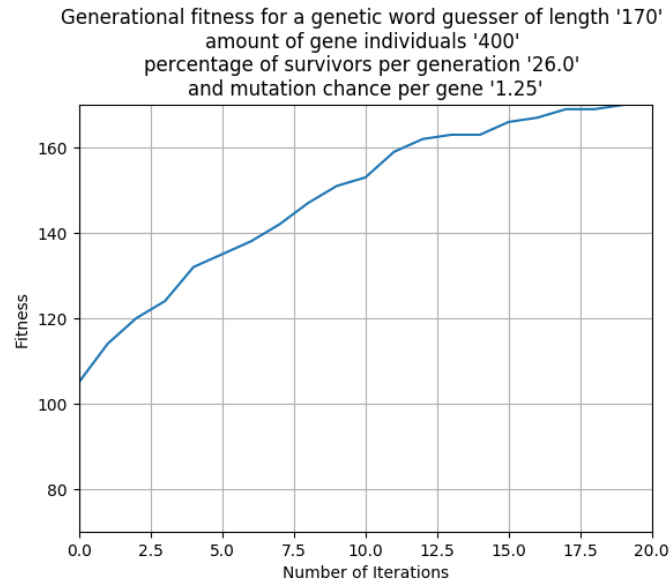


Figure 64: Test using binary alphabet for Mutation Chance 1

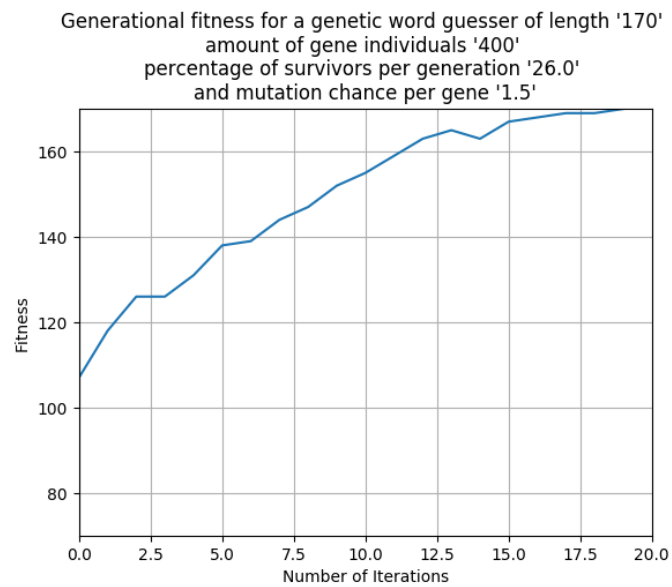


Figure 65: Test using binary alphabet for Mutation Chance 1

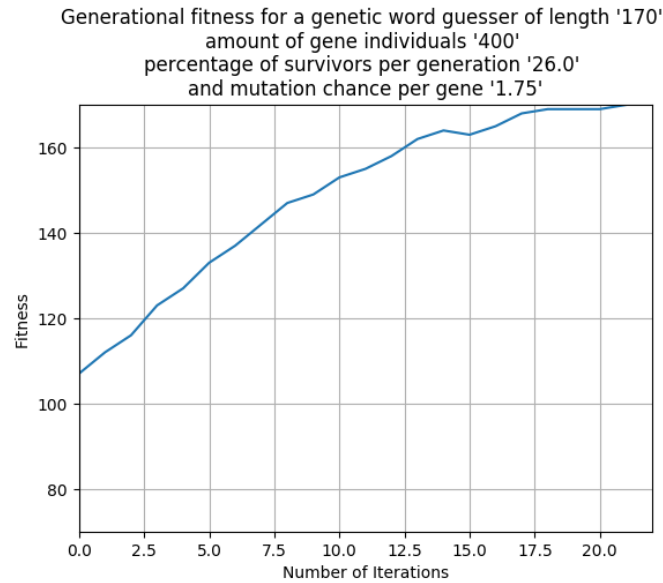


Figure 66: Test using binary alphabet for Mutation Chance 1

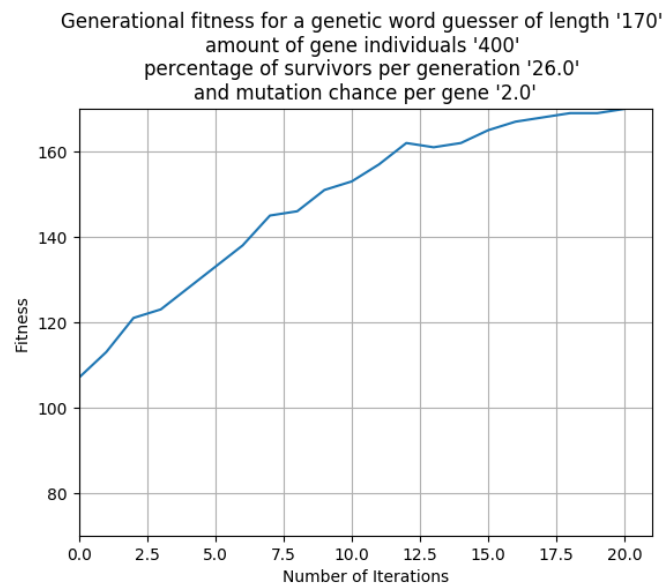


Figure 67: Test using binary alphabet for Mutation Chance 2

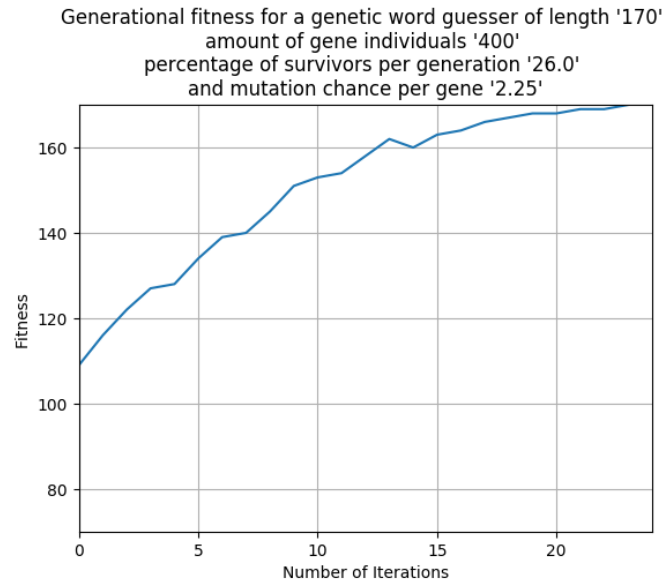


Figure 68: Test using binary alphabet for Mutation Chance 2

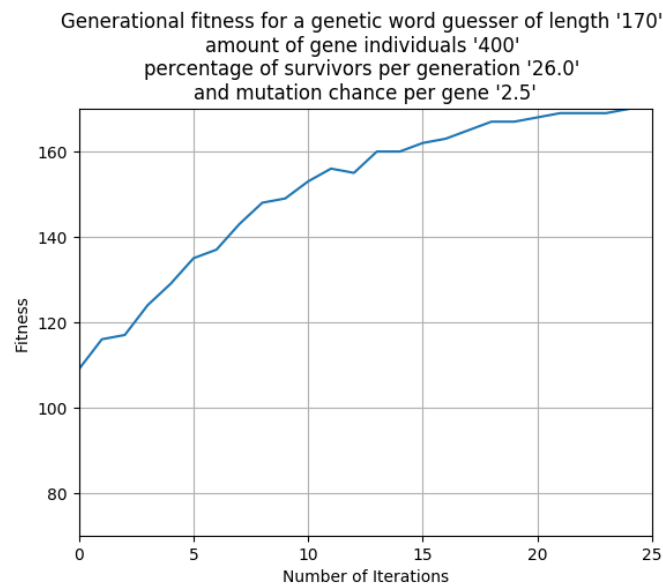


Figure 69: Test using binary alphabet for Mutation Chance 2

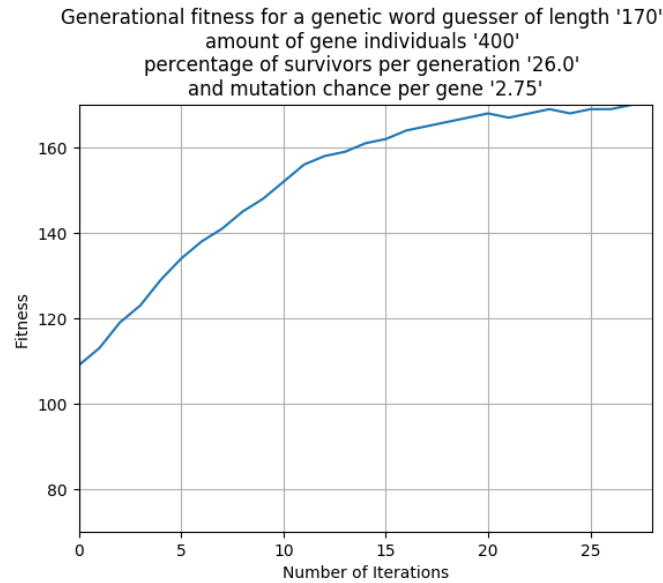


Figure 70: Test using binary alphabet for Mutation Chance 2

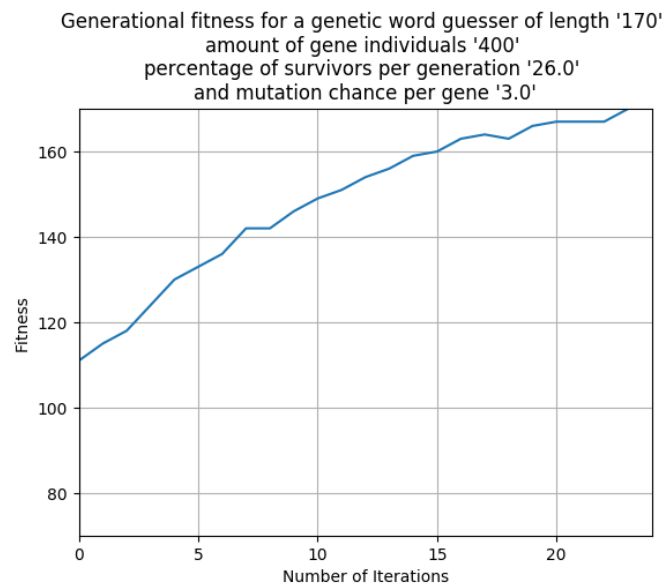


Figure 71: Test using binary alphabet for Mutation Chance 3

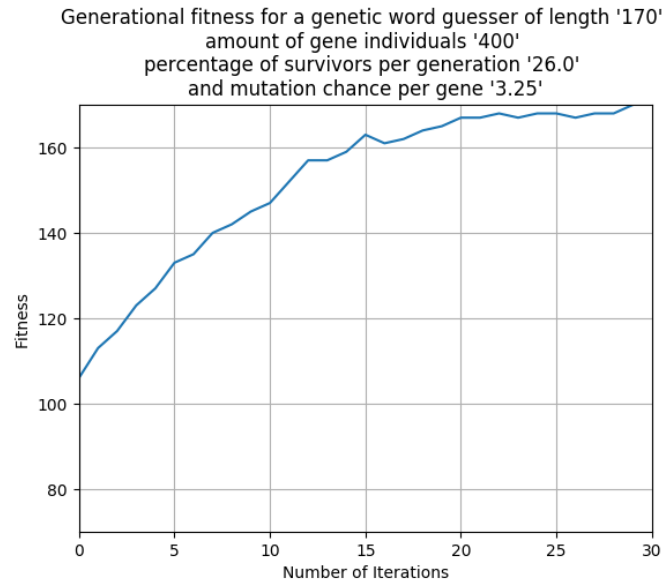


Figure 72: Test using binary alphabet for Mutation Chance 3

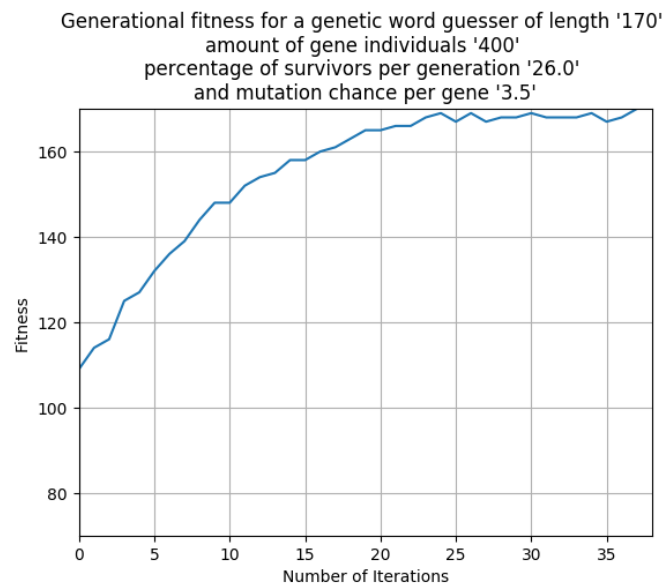


Figure 73: Test using binary alphabet for Mutation Chance 3

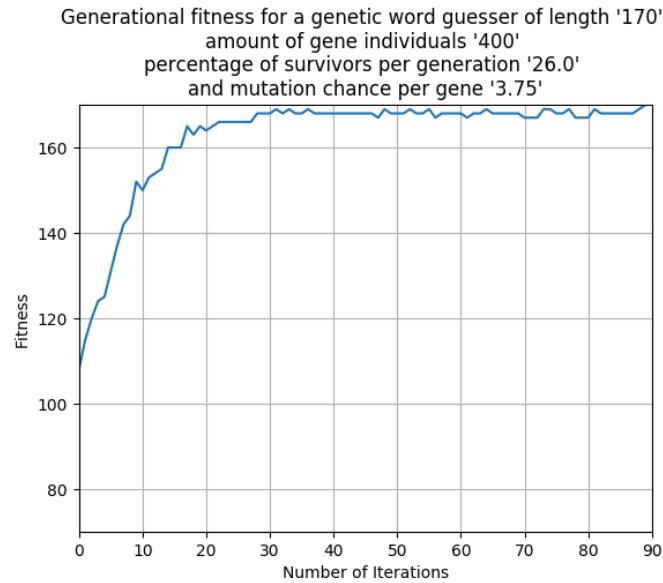


Figure 74: Test using binary alphabet for Mutation Chance 3

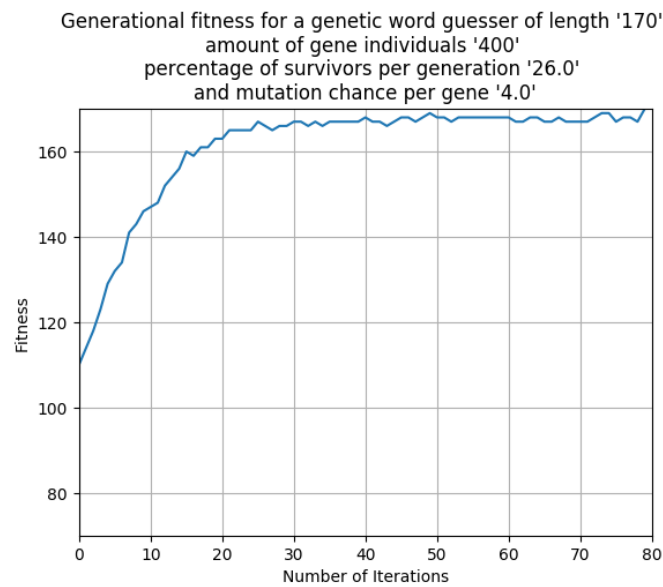


Figure 75: Test using binary alphabet for Mutation Chance 4

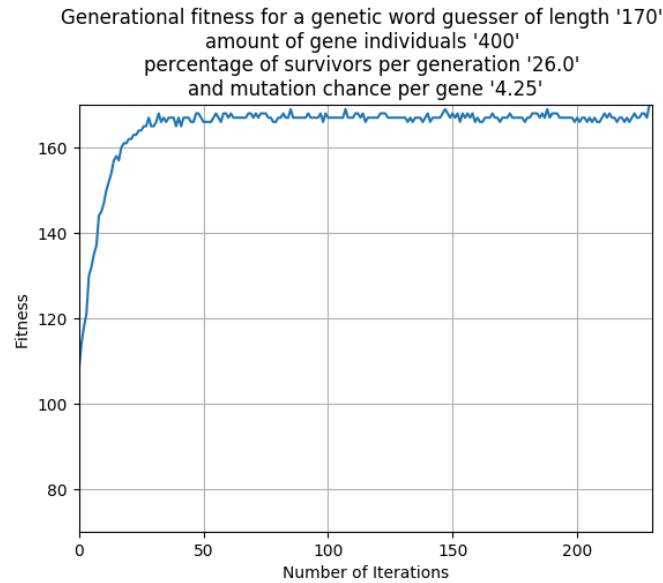


Figure 76: Test using binary alphabet for Mutation Chance 4

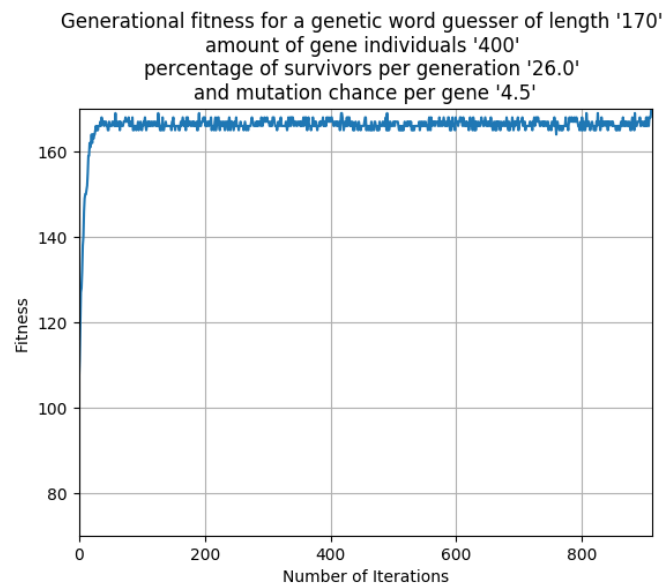


Figure 77: Test using binary alphabet for Mutation Chance 4

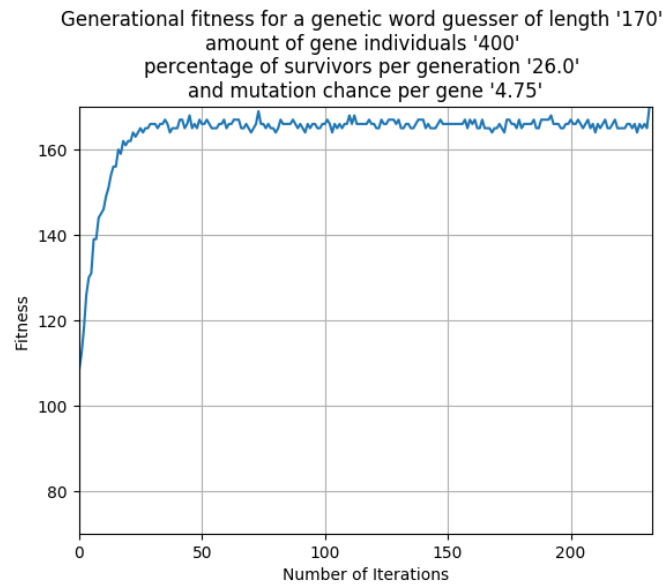


Figure 78: Test using binary alphabet for Mutation Chance 4

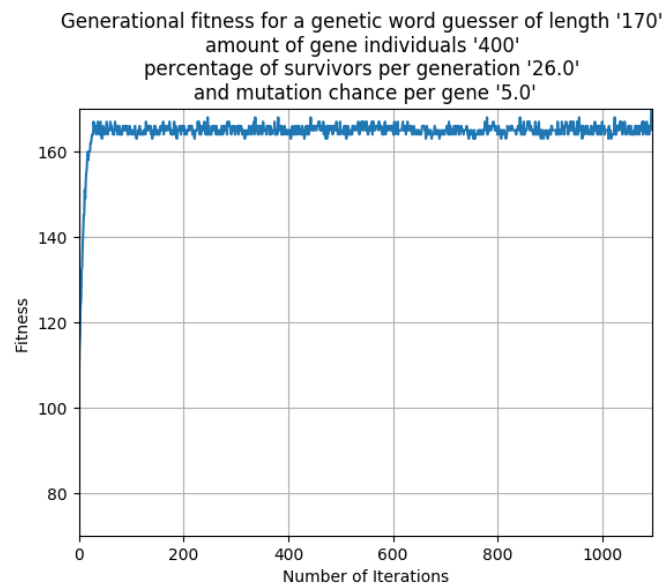


Figure 79: Test using binary alphabet for Mutation Chance 5

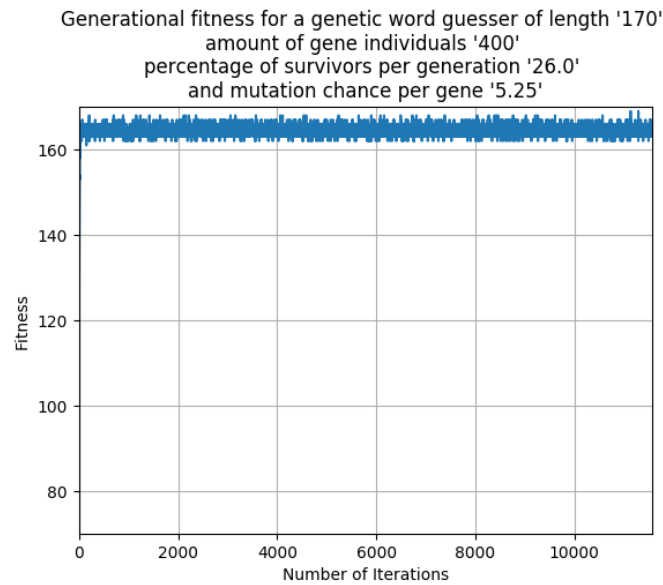


Figure 80: Test using binary alphabet for Mutation Chance 5

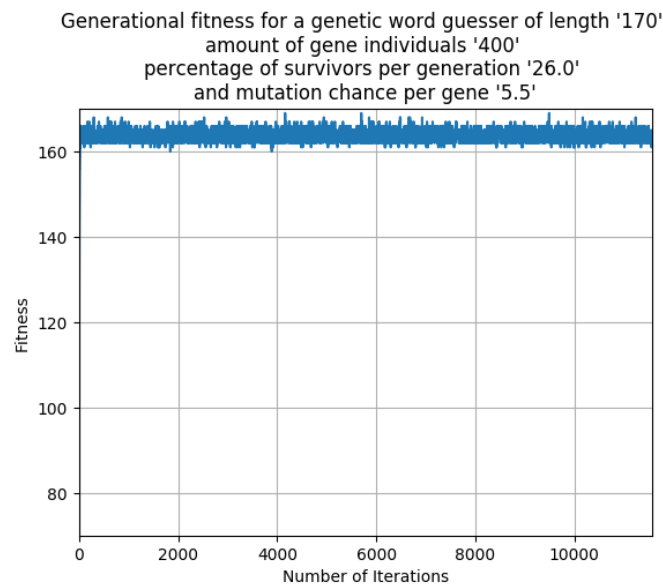


Figure 81: Test using binary alphabet for Mutation Chance 5

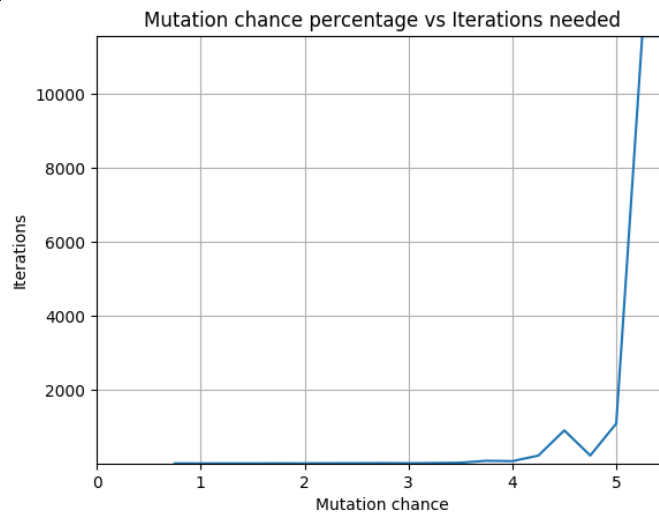


Figure 82: All mutation tests for binary alphabet

6.1.5 English alphabet test

6.2 N-Queens

6.2.1 Board Configurations

Solution to the 4-Queens Problem after 1 Attempts, 149 Iterations

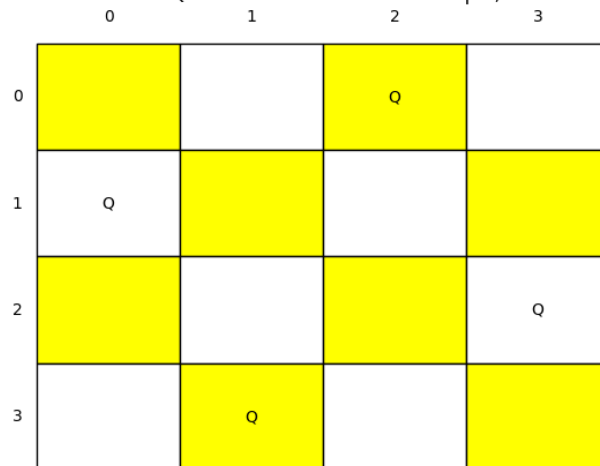


Figure 83: Result of the N-Queen Problem for N 4

Solution to the 8-Queens Problem after 1 Attempts, 323 Iterations

	0	1	2	3	4	5	6	7
0								Q
1				Q				
2	Q							
3			Q					
4						Q		
5		Q						
6							Q	
7					Q			

Figure 84: Result of the N-Queen Problem for N 8

Solution to the 12-Queens Problem after 2 Attempts, 1499 Iterations

	0	1	2	3	4	5	6	7	8	9	10	11
0				Q								
1						Q						
2										Q		
3							Q					
4	Q											
5				Q								
6								Q				
7		Q										
8									Q			
9			Q									
10						Q						
11											Q	

Figure 85: Result of the N-Queen Problem for N 12

Solution to the 16-Queens Problem after 3 Attempts, 3226 Iterations

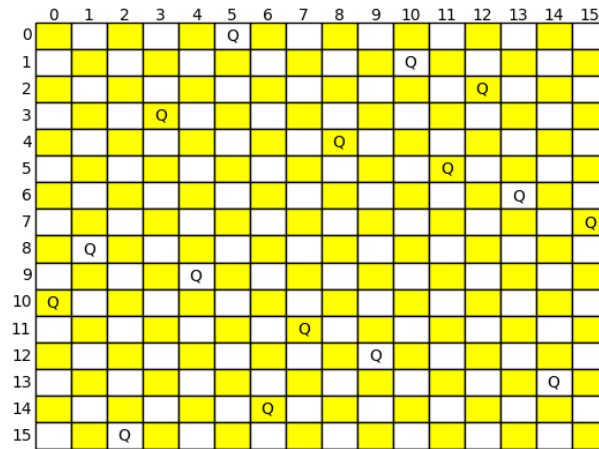


Figure 86: Result of the N-Queen Problem for N 16

Solution to the 20-Queens Problem after 4 Attempts, 6119 Iterations

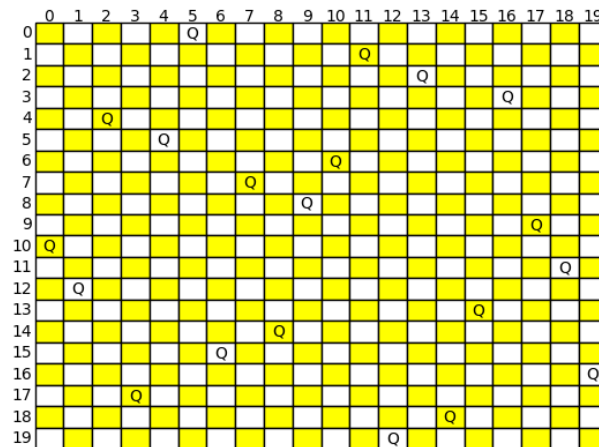


Figure 87: Result of the N-Queen Problem for N 20

Solution to the 24-Queens Problem after 30 Attempts, 56033 Iterations

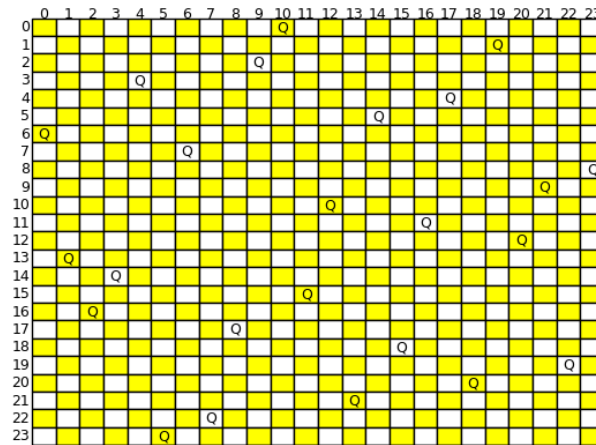


Figure 88: Result of the N-Queen Problem for N 24

Solution to the 28-Queens Problem after 6 Attempts, 16784 Iterations

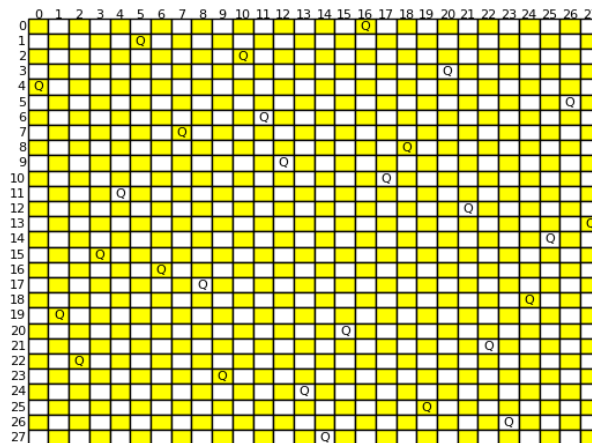


Figure 89: Result of the N-Queen Problem for N 28

Solution to the 32-Queens Problem after 4 Attempts, 13185 Iterations

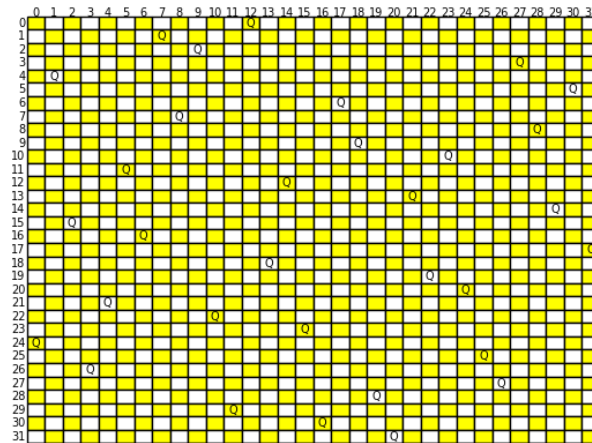


Figure 90: Result of the N-Queen Problem for N 32

Solution to the 36-Queens Problem after 12 Attempts, 50945 Iterations

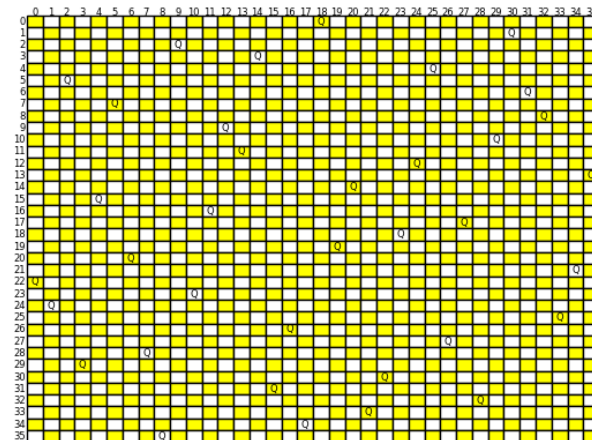


Figure 91: Result of the N-Queen Problem for N 36

Solution to the 40-Queens Problem after 3 Attempts, 17453 Iterations

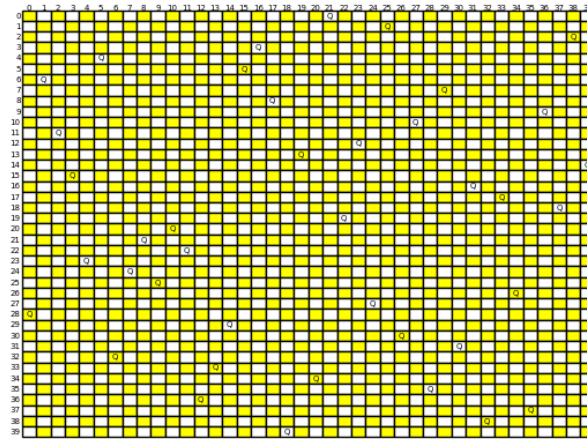


Figure 92: Result of the N-Queen Problem for N 40

6.2.2 Fitness Plots

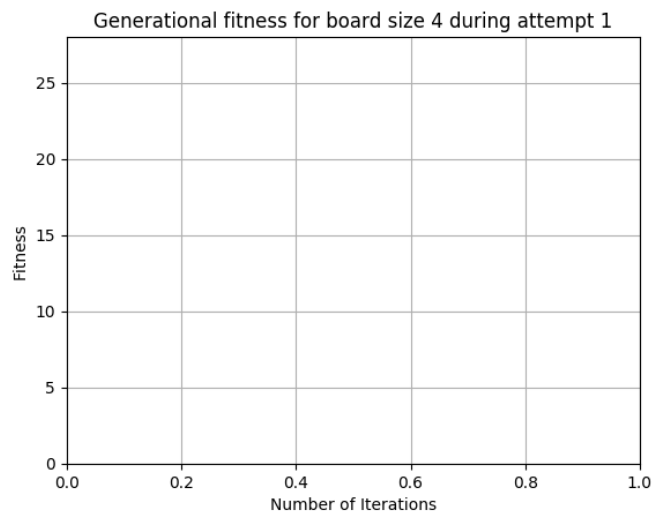


Figure 93: Fitness plot of the N-Queen Problem for N 4

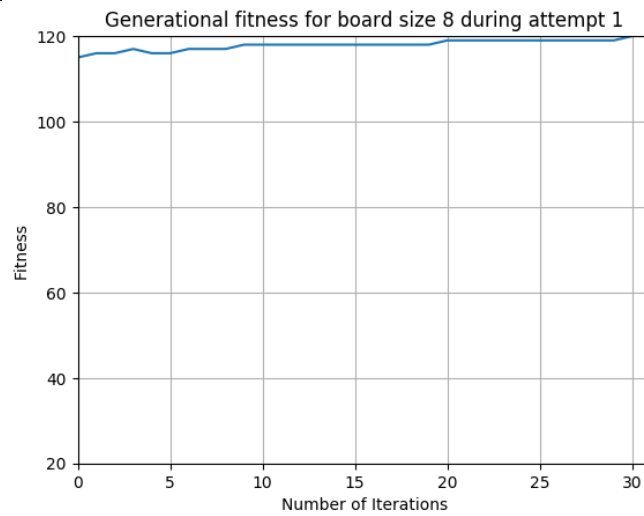


Figure 94: Fitness plot of the N-Queen Problem for N 8

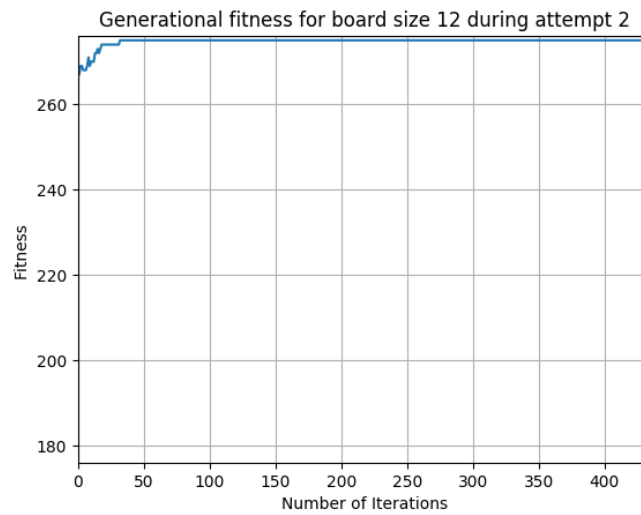


Figure 95: Fitness plot of the N-Queen Problem for N 12

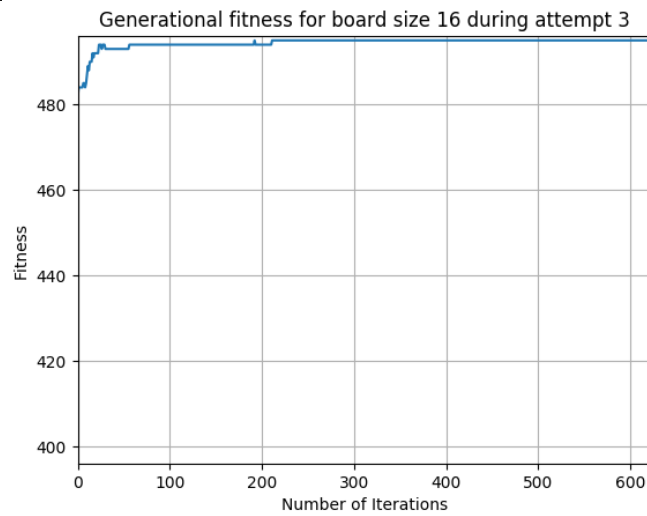


Figure 96: Fitness plot of the N-Queen Problem for N 16

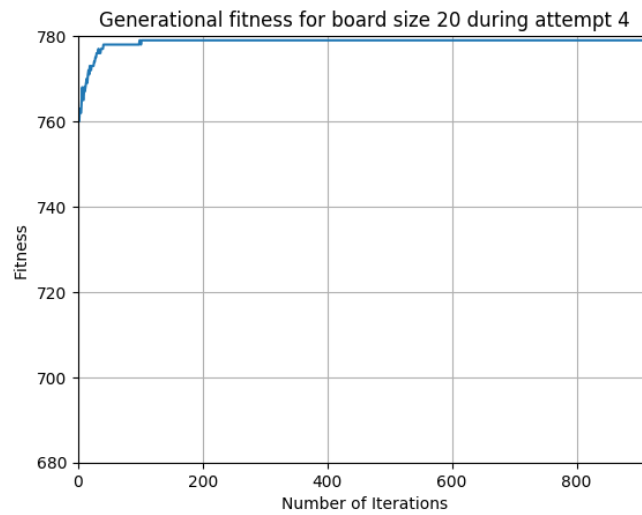


Figure 97: Fitness plot of the N-Queen Problem for N 20

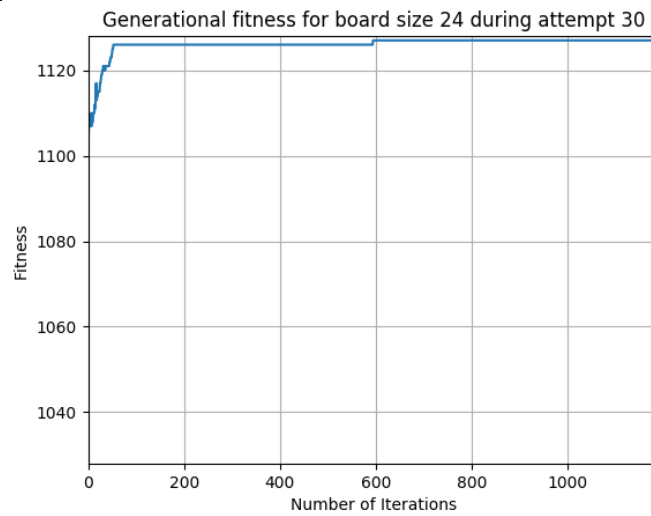


Figure 98: Fitness plot of the N-Queen Problem for N 24

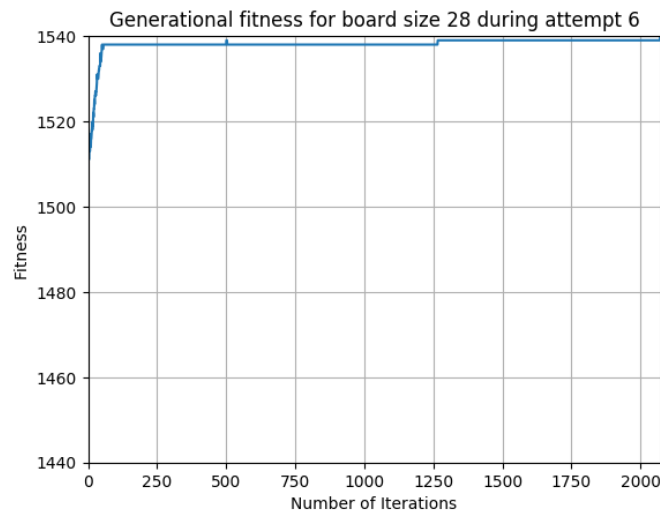


Figure 99: Fitness plot of the N-Queen Problem for N 28

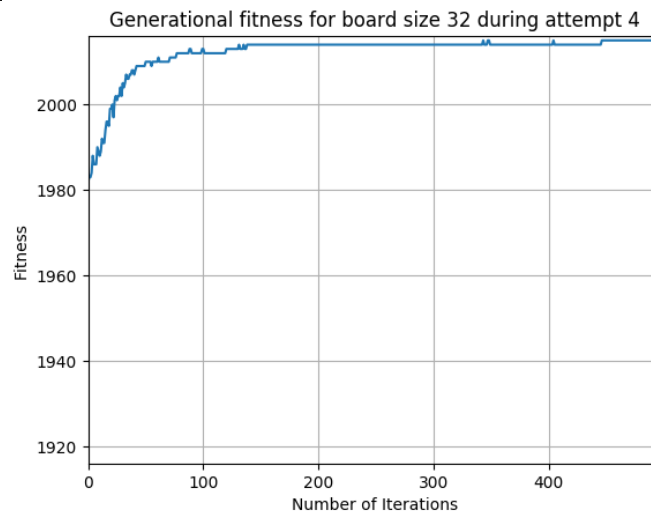


Figure 100: Fitness plot of the N-Queen Problem for N 32

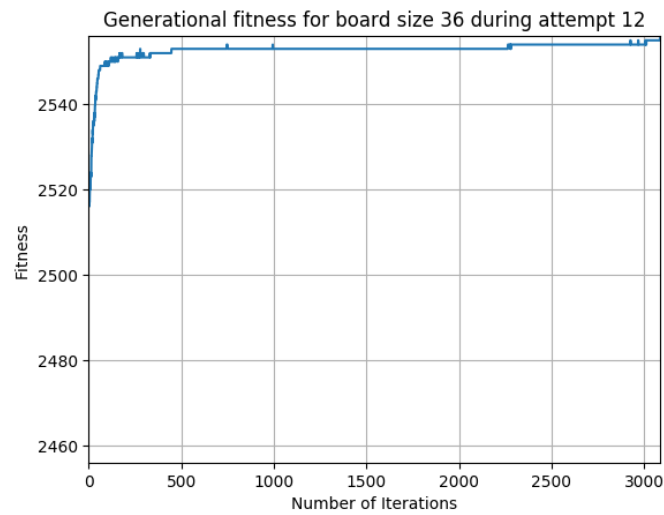


Figure 101: Fitness plot of the N-Queen Problem for N 36

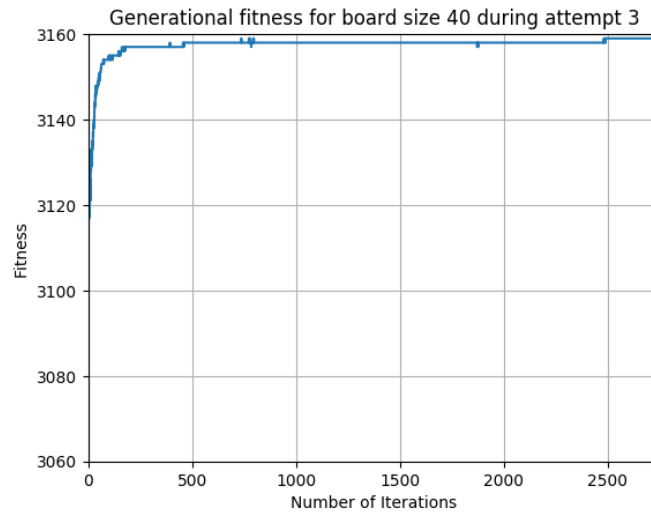


Figure 102: Fitness plot of the N-Queen Problem for N 40