Optimizing Keyboard Layouts Using AI

Author: Luke Cardona
ID: 11803H

1. Introduction:

The traditional Qwerty and Azerty keyboard layouts are known to have high finger movement and could be optimized. Genetic Algorithms has been selected as a technique to optimize these keyboards. The aim of this paper is to discuss different genetic algorithms approaches and find the approach best fit to solve this problem. The aim of this study is to develop an optimal keyboard using genetic algorithms and compare the efficiency of the new keyboards with the traditional Qwerty keyboard layout and Azerty keyboard layout. The study compares results of different genetic algorithm properties in an attempt to find the best genetic algorithm properties that will produce the most optimal keyboard.

2. Building the data:

Keyboard Characters

The keyboard was composed of the 26 English letters and the following four characters: '.' ',' ';' '?'. These characters were chosen as they are the first 10 characters on a Qwerty keyboard per row.

Training dataset

The dataset of the genetic algorithm was built from 4 books of "Project Gutenberg" [1-4] and is composed of approximately 1 million characters. The books were downloaded as text files and combined into a single text file "unrefinedDataset.txt". The dataset was then filtered from characters not present on the keyboard and put into a file called "refinedDataset.txt" which was used for all the GA's. Up to the first 200,000 characters of the dataset were used to train the genetic algorithms.

Sample Dataset

The characters not used to train the GA found in the file called "refineDataset.txt", this created a sample dataset of around 800,000 characters to test the GA's keyboards with. This is being done

to remove any bias the GA has towards the training data and instead using a dataset which the GA has not been trained on.

3. Building the GAs

3.1 Finger rules

The person operating this keyboard is assumed to use all 10 fingers to type. When using the 10 fingers to type, the two thumbs are assumed to hover the spacebar and never move off the spacebar. Thus these two fingers will always have a distance of zero and therefore can be ignored when calculating the fitness score of a keyboard. As a result the empty spaces in the dataset were also removed. The remaining starting finger positions would all be in the middle row with 4 fingers on the 4 most left keys of the row and the same with the right. Refer to figure 1. This layout mimics a layout put forward by adumb[5].

Figure 1: Starting finger positions displayed on a qwerty keyboard

Each finger has an area that it is responsible for. Therefore the keyboard has been divided into 8 sections. Each finger has limited movement to its specific section and cannot cross into another's finger sections. The section division also mimics the one present by adumb[5].

Figure 2: Visual representation of each finger's section and their possible movement.

The fingers will reset to their starting positions when a different finger is used to type the next character. Using Figure 3 below, the fingers would start in their normal starting position. The finger on the 'F' key will move to 'T' then 'B'. Since the next letter E is in a position which the previous finger could not reach, the fingers are reset to their starting positions. Then the finger on the key 'D' is moved to 'E'. Since 'R' is a position which the previous finger could not reach, the finger positions are reset to their starting positions. Then the finger is moved from the key 'F' to key 'R'.

Figure 3: Display the fingers movement for the string sequence "TBER". Red arrows represent calculated distance, yellow arrows represent uncalculated distance.

The finger resetting tries to achieve the most optimal finger positions for the longest possible period. Thus the 8 most common used letters in the GA will gradually move to the starting finger positions.

3.2 Fitness Calculation

The fitness of each keyboard is calculated by getting the geological euclidean distance between the finger location and the key that needs to be pressed. The key's geological locations are stored in a dictionary. The fitness of a keyboard is the $Total\ Finger\ Distance/Length_{\text{String}}$. This makes the score independent of the number of characters being used in the dataset.

3.3 Population Creation

The population for the GA would be created using randomly generated keyboards. This method was selected as random generation allows for a diverse starting population.

3.4 Selection Methods

The selection methods determine which keyboards are selected to be parents of the next generation's keyboard. Two selection methods are used for this genetic algorithm:

1. **Best 50% selection:** This breeding method will always select the 50% best fit keyboards to produce the next generation of keyboards. This selection method quickly eliminates all low scoring keyboards. The GA then randomly chooses two unique parents from these 50 keyboards until the next generation is fully generated.

2. **Rank Based Selection:** This breeding method will choose 50 keyboards randomly based on their fitness. It is a form of modified roulette wheel selection [6]. The keyboards are sorted by their fitness and given a probability of being selected as parents. The probability difference between two adjacent keyboards is a fixed value. The GA then randomly chooses two unique parents from these 50 keyboards until the next generation is fully generated.

3.5 Crossover Functions

The crossover functions used by this GA split the keyboard vertically by one or more points and then take key positions from both parents. Each crossover function takes 2 parents and will produce two children. This GA makes these two Crossover methods:

1. **Single Point Crossover** - A single random point is chosen on the keyboard and is then used to split the keyboard into two parts. The children inherit from the first key to the crossover point from their respective parents. The remaining keys are inherited from the other parent by traversing the keyboard vertically and adding all keys which are not present in the child already. [7]

2. **Two Point Crossover** - Two random points are chosen on the keyboard and then the keyboard is split into three parts. The children inherit from the first key to the crossover points from their respective parents. After the first crossover point the children start to inherit keys from the other parent until they reach a length of the second crossover point. Then remaining keys are inherited from the original parent of the children[7].

3.6 Elitism

The GA used an Elitism rate of 10%. The top 10% scoring keyboards from each population would be passed on to the next generation of keyboards.

3.7 Mutations

Mutations randomly change the keyboards to prevent the GA from getting stuck in a local minima. The mutation was applied on a Keyboard, with mutation rates from 10%-20% (depending on the GA). The GA's used three methods for mutations [8]:

1. **Swap Mutation**: Two random keys would randomly swapped positions on the keyboard.
2. **Shuffle Mutation**: Two random points on the keyboard would be randomly selected and the key positions in between those points would be scrambled .
3. **Inverse Mutation:** Two random points on the keyboard would be randomly selected and the key positions would become inverted on the keyboard.

4. Genetic Algorithms Keyboard Design :

Expected Results

For the purpose of this paper, 5 different Test cases were initially created with different traits, with the order of expected performance from worst to best performance being the following:

1. **Test Case 1:** Basic GA
2. **Test Case 2:** Rank GA
3. **Test Case 3:** Elitism + Rank GA
4. **Test Case 4:** Two Point Crossover + Elitism + Rank GA
5. **Test Case 4:** Final GA (Test Case 4 + Mutation)

The worst performing should be the Basic GA since it is expected to converge quickly and thus would produce the lowest score.

Adding rank based selection to the GA will allow lower scoring keyboards to be selected as parents. This will increase the number of generations needed to converge and add variety to the population.

Adding Elitism should make the GA converge quicker and should improve the overall fitness of the keyboards by guaranteeing that the best keyboards are passed on from generation to generation.

Two point Crossover should increase the variety of keyboards that are created from children and thus will take longer to coverage but it will also improve the overall fitness of the keyboards.

The best performer should be the Final GA since it combines different GA optimization techniques.

The best keyboard should have the eight most frequent letters used in the English alphabet on top of the eight starting positions. With the least used letters being in the middle section of the keyboard.

Qwerty and Azerty Layouts

The Qwerty and Azerty layouts fitness was calculated using the sample dataset being used to test the fitness of the GA's with Azery scoring 0.8988 and Qwerty scoring 0.8322.

First Evaluation of GA's

Test Case 1: Basic GA: The GA has the following traits:

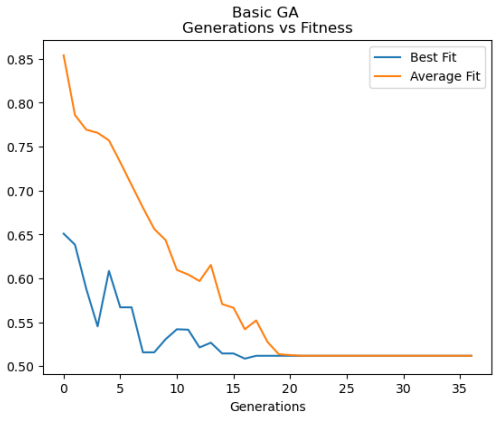
- Population: 50
- Generations: 50
- Text Corpus: 100,000 characters
- Crossover Method: Single point crossover
- Selection Model: Best 50%
- Mutation Method: None
- Elitism: None
- Stopping Mechanism: 15 generations of no change or 50 generations

Results

Basic GA details	Run 1	Run 2	Average
Generations Reached	36	34	35
CPU Time run $(seconds)$	580.47	523.98	552.22
Best Fitness Score $(FingerDistance\ /Character)$	0.5226	0.5684	0.5455

Visual Representations:

Run 1



Run 2

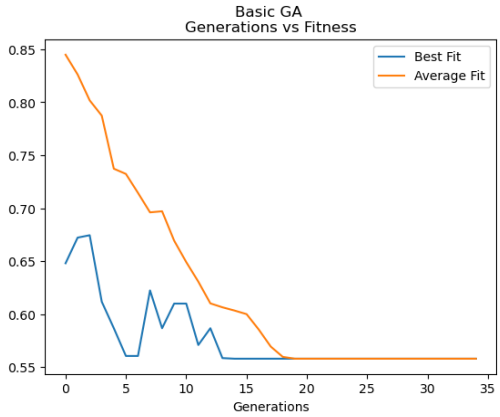


Figure 4: Visual representation of the fitness scores against the number of the generations for both runs of test case 1.

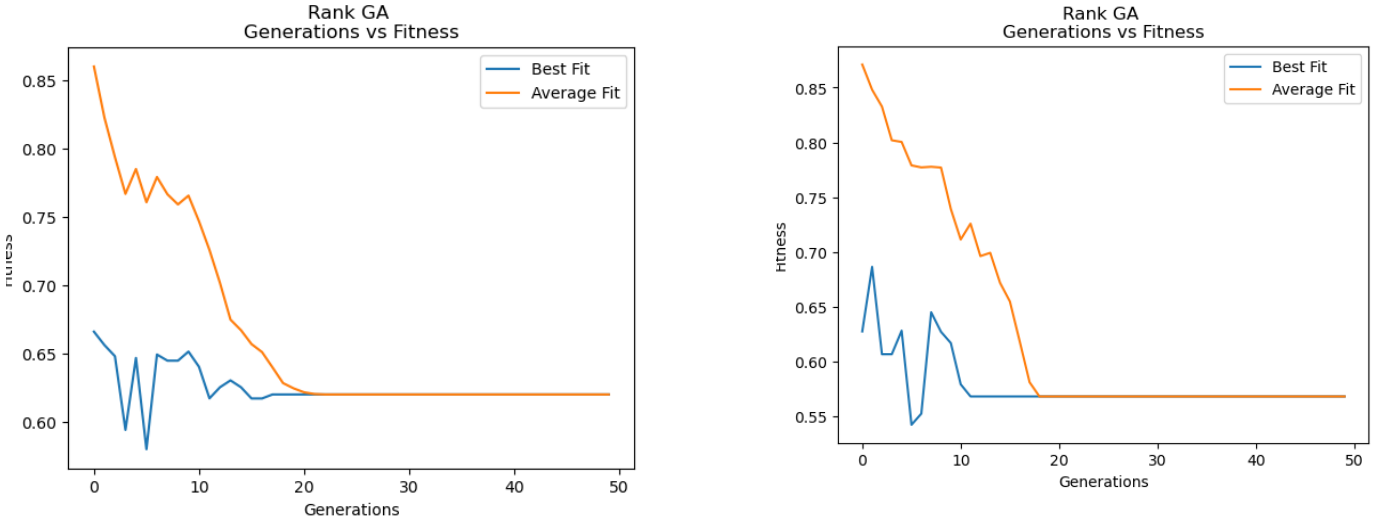
Test Case 2: Rank GA: The GA has the following traits:

- Generations: 50
- Population: 50
- Text Corpus: 100,000 characters
- Crossover Method: Single point crossover
- Selection Model: Rank selection
- Mutation Method: None
- Elitism: None
- Stopping Condition: 50 generations

Results

Rank GA details	Run 1	Run 2	Average
CPU Time run $(seconds)$	796.53	763.23	779.88
Best Fitness Score $(FingerDistance\ /Character)$	0.6196	0.5738	0.5967

Visual Representations:



Run 1 ***Run 2***

Figure 5: Visual representation of the fitness scores against the number of the generations for both runs of test case 2

Test Case 3: Elitism + Rank GA: The GA has the following traits:

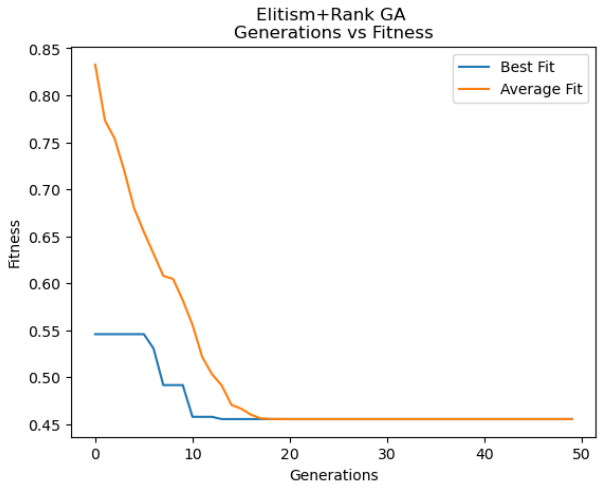
- Generations: 50
- Population: 50
- Text Corpus: 100,000 characters
- Crossover Method: Single point crossover
- Selection Model: Rank selection
- Mutation Method: None
- Elitism: 10%
- Stopping Condition: 50 generations

Results:

Elistim + Rank GA details	Run 1	Run 2	Average
CPU Time run $(seconds)$	716.73	736.03	726.38
Best Fitness Score $(FingerDistance\ /Character)$	0.4588	0.4904	0.4746

Visual Representations:

Run 1



Run 2

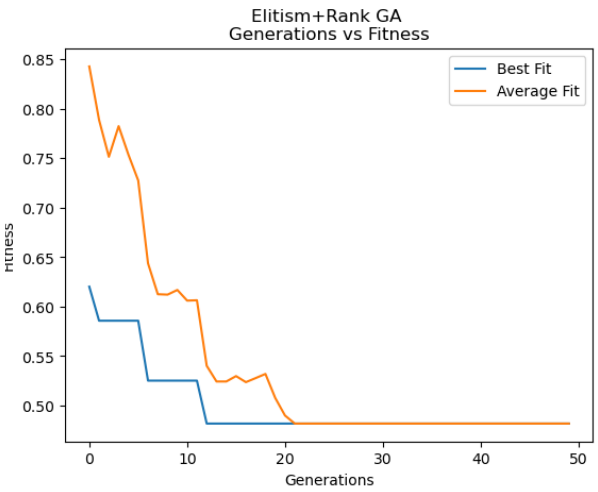


Figure 6: Visual representation of the fitness scores against the number of the generations for both runs of test case 3

Test Case 4: Two Point Crossover + Elitism + Rank GA: The GA has the following traits

- Generations: 50
- Population: 50
- Text Corpus: 100,000 characters
- Crossover Method: Two point crossover
- Selection Model: Rank selection
- Mutation Method: None
- Elitism: 10%
- Stopping Condition: 50 generations

Results:

Two point crossover + Elistim + Rank GA details	Run 1	Run 2	Average
CPU Time run $(seconds)$	426.03	432.80	429.42
Best Fitness Score $(FingerDistance\ /Character)$	0.5208	0.4737	0.4972

Visual Representations:

Run 1 ***Run 2***

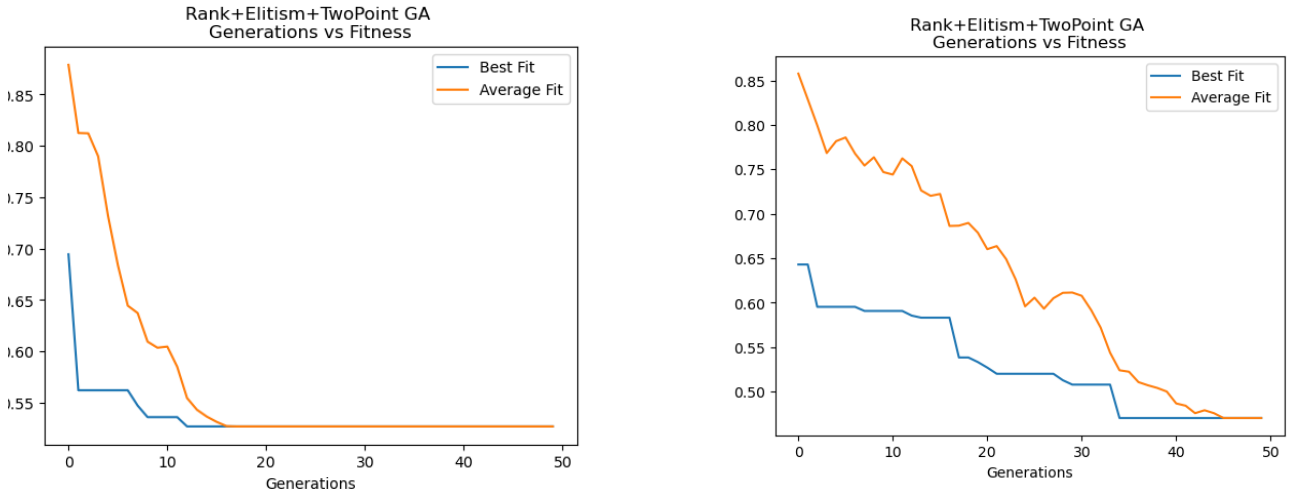


Figure 7: Visual representation of the fitness scores against the number of the generations for both runs of test case 4

Test Case 5: Final GA. The GA has the following traits:

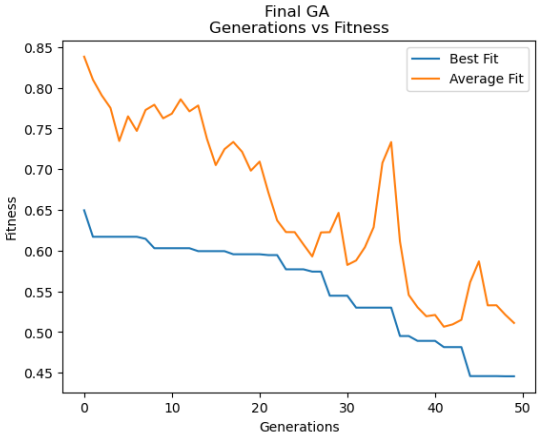
- Generations: 50
- Population: 50
- Text Corpus: 100,000 characters
- Crossover Method: Two point crossover
- Selection Model: Rank selection
- Mutation Rate: 10%
- Mutation Methods: Swap, Shuffle, Inverse: 33.33%
- Elitism: 10%
- Stopping Condition: 50 generations

Results:

Final GA details	Run 1	Run 2	Average
CPU Time run $(seconds)$	764.14	738.48	751.31
Best Fitness Score $(FingerDistance\ /Character)$	0.4462	0.4709	0.4585

Visual Representations:

Run 1



Run 2

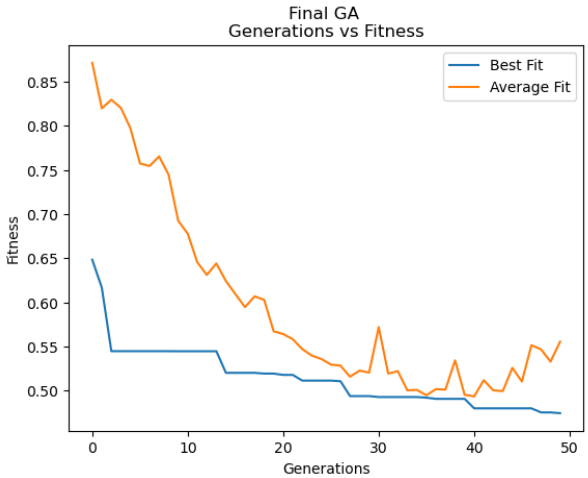


Figure 8: Visual representation of the fitness scores against the number of the generations for both runs of test case 5

Evaluation of Test Cases

From the test cases we saw that they GA's performed in the following order (best to worst) when comparing their average best fitness scores:

1. Final GA
2. Elitism + Rank GA
3. Two Point Crossover + Elitism + Rank GA
4. Basic GA
5. Rank GA

Test Case 1: The basic GA is the base case algorithm which is used to evaluate the other test cases performance. The basic GA's best average keyboard performance was already an improvement from the Qwerty layout (34.4% improvement) and Azerty layout (39.3% improvement). With an average best fitness score of 0.5455 and the best run performing at 0.5226. The algorithm also took 552s of dedicated CPU time to execute. This algorithm has a unique stopping condition which stopped the GA after 15 generations of no change and therefore ended the GA at around the 35th generation. This stopping condition is not present in the other test cases, thus the time for this GA to execute is expected to be less than the rest of the test cases.

Test Case 2: The rank GA performed worse than the basic GA, with an average fitness score of 0.5967, a 9.39% increase in the fitness score. It was the worst performer from all the test cases, due to test case 2 being allowed to choose the same parent more than once for the selection population. This occurred due to the introduction of rank based selection. Therefore the parents with the highest probability were getting selected the most, decreasing the variety of parents for the next generation of keyboards. This caused test case 2 to converge earlier, as can be seen in Figure 5. Therefore the basic GA outperformed the rank GA by 9.4% when comparing the average best fitness scores. Test case 2 takes 779.88s of dedicated CPU time to run which is higher than test case 1 time due to the different stopping conditions.

Test Case 3: Test case 3 had an average fitness score 0.4746. Introducing elitism saw a significant improvement over the average best fitness scores of test case 1 and test case 2, with a 14.94% and 25.73% improvement respectively. Furthermore as can be seen in Figure 6 the most notable improvement is that the best keyboard never goes up in fitness score. Introducing elitism ensured that the population's best keyboards survived and moved on. The elitism converged around the same point as test case 2, as can be seen by comparing Figure 5 and Figure 6. Therefore it can be concluded that the elitism rate is not too high, as to have a significant impact on the algorithm's converging point. Test case 3 takes an average of 726.38s of dedicated CPU time to run. This is lower than test case 2, since introducing Elitism leads to less computation

needed by the genetic algorithm. Moreover this can be supported by looking at the time percentage difference between test case 2 and test case 3. Test case 3 had a 7.37% decrease in time from test case 2, which is similar to the 10% elistim rate used for test case 3. Therefore due to the similarities in percentage one can conclude that the time difference is due to the change in elitism rate.

Test Case 4: Test case 4 had an average fitness score of 0.4972. Changing the crossover method to a two point crossover saw a slight increase (4.54%) in the average best fitness score when compared to test case 3. This means that the two point cross over did not benefit the algorithm. This is likely due to the increase in the variety of keyboards between generations. Figure 7 second run supports this claim since from all the test cases from one to four, test case four has been the slowest to converge. Moreover two anomalies are also noted in this test case. Due to an error, this test case was re-run separately. At the time of running this algorithm the CPU was under less load, resulting in less time to execute than the other algorithms. The other anomaly is that run 1 has converged quickly. Due to the nature of randomly generated populations, this could be a scenario where the genetic algorithm started with a bad population, resulting in the quick conversion.

Test Case 5: Test case 5 had an average fitness score of 0.4585. The addition of mutation to the genetic algorithm has seemed to greatly assist the performance of the genetic algorithm. The algorithm performed 7.78% better than test case 4 and 3.39% better than test case 3 The effects of mutation can be clearly seen by looking at the irregular line of the average fit in Figure 8 in both runs. Adding mutation also prevented the algorithm from ever converting and therefore the best keyboard could potentially be further improved if the algorithm could have run for more generations. The time increase to process mutation was small compared to Test case 3 but due to the anomaly of Test case 4, the definitive effect of mutation on the algorithm's time to execute cannot be determined since two point crossover was also introduced.

Improvements

Due to test case 5 being the best performer it will be used as the base case for improving the GA by tweaking parameters. Four more test cases will be run to tweak the parameters of the GA.

All of the previous test cases used a dataset of 100,000 characters, the first potential improvement involves determining the difference in the fitness score, if the dataset is reduced to 10,000 characters. This test case was conducted to test if it is possible to significantly decrease the speed by which the genetic algorithm can run without affecting the results of the test cases. The results of the fitness score should vary minimally and the algorithm should finish around 90% faster.

Test Case 6: Low Char GA. The GA has the following traits:

- Generations: 50
- Population: 50
- Text Corpus: 10,000 characters
- Crossover Method: Two point crossover
- Selection Model: Rank selection
- Mutation Rate: 10%
- Mutation Method:Swap, Shuffle, Inverse: 33.33%
- Elitism: 10%
- Stopping Condition: 50 generations

Results:

Reduced dataset size GA details	Run 1	Run 2	Average
CPU Time run $(seconds)$	80.4	77.06	78.3
Best Fitness Score $(FingerDistance\ /Character)$	0.4511	0.4185	0.4348

Visualizations:

Run 1 ***Run 2***

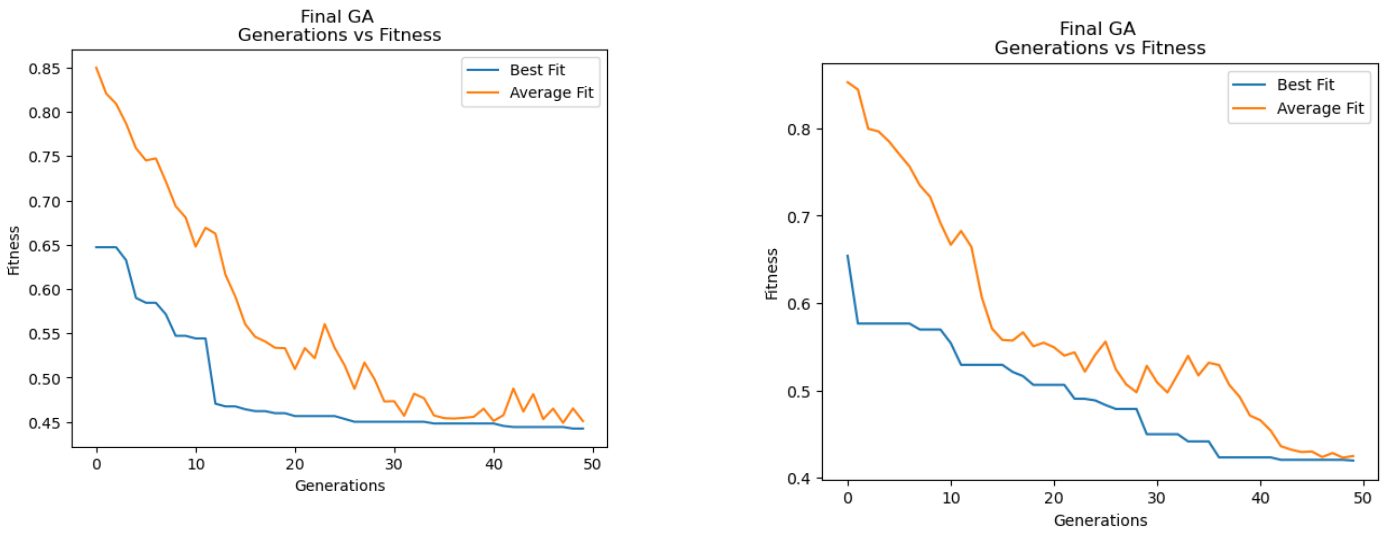


Figure 9: Visual representation of the fitness scores against the number of the generations for both runs of test case 6

Test Case 6 Evaluation: Test case 6 had an average time execution of 78.73s which is a drastic change over all the other test cases. It had an 89.52 % decrease in the amount of time taken to execute. The average fitness score of test case 6 is 0.4348 which shows that a reduction in the dataset size did not affect the fitness score. Moreover it is notable that there was an improvement in the average keyboard fitness of test case 6 when compared to test case 5. This improvement can be allocated to the random probability of mutation and the starting population being randomly generated. Due to the significant decrease in time of execution, the new dataset size was set to 10,000 for the future test cases.

One of the parameters that had not been tested was the population size. The population size for test case 7 was set to 100 to test the effect of doubling the population. It is predicted that the increase in population should double the amount of time the GA needs to execute with some slight improvements to the best fitness score.

Test Case 7: 100 Population GA. The GA has the following traits:

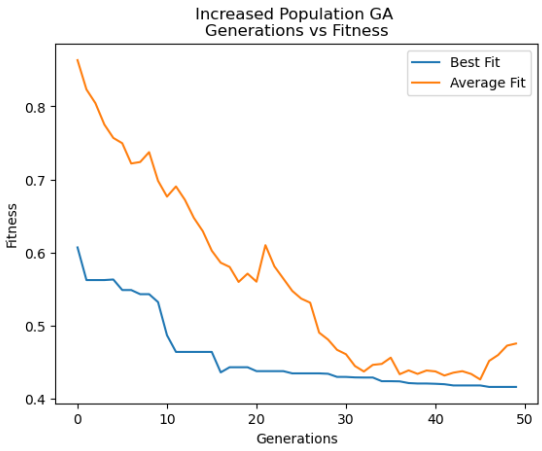
- Generations: 50
- Population: 100
- Text Corpus: 10,000 characters
- Crossover Method: Two point crossover
- Selection Model: Rank selection
- Mutation Rate: 10%
- Mutation Method:Swap, Shuffle, Inverse: 33.33%
- Elitism: 10%
- Stopping Condition: 50 generations

Results:

100 Population GA details	Run 1	Run 2	Average
CPU Time run $(seconds)$	158.97	161.90	160.44
Best Fitness Score $(FingerDistance\ /Character)$	0.4168	0.4591	0.4378

Visualizations:

Run 1



Run 2

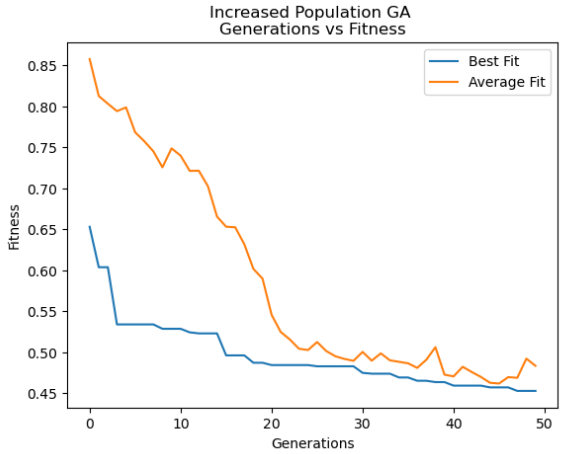


Figure 10: Visual representation of the fitness scores against the number of the generations for both runs of test case 7

Test Case 7 evaluation: Test case 7 had an average fitness score of 0.4378. The increase in population size had minimal effect on the fitness score of test case 7, with a 0.69% decrease in the average best fitness score. There was a 104% time increase from test case 6 to test case 7. This is due to the doubling in population, since twice as many keyboards need to be computed for each generation. Therefore it can be concluded that the doubling of the population does not significantly improve the genetic algorithm's fitness performance while greatly decreasing it's efficiency.

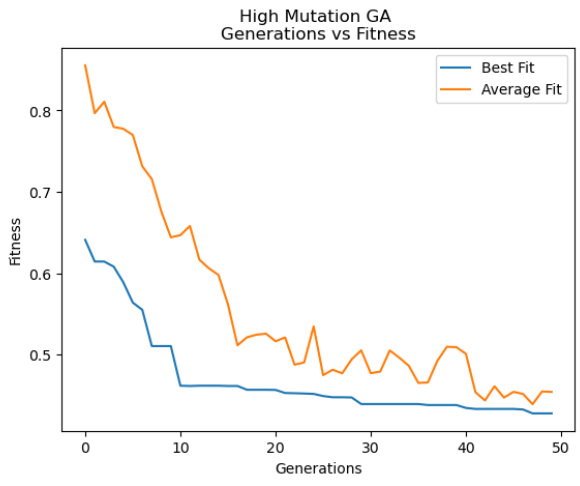
Test Case 8: High mutation GA. The GA has the following traits:

- Generations: 50
- Population: 50
- Text Corpus: 10,000 characters
- Crossover Method: Two point crossover
- Selection Model: Rank selection
- Mutation Rate: 20%
- Mutation Method:Swap, Shuffle, Inverse: 33.33%
- Elitism: 10%
- Stopping Condition: 50 generations

High mutation GA details	Run 1	Run 2	Average
CPU Time run $(seconds)$	38.86	40.12	39.49
Best Fitness Score $(FingerDistance\ /Character)$	0.4257	0.4306	0.4281

Visualizations:

Run 1



Run 2

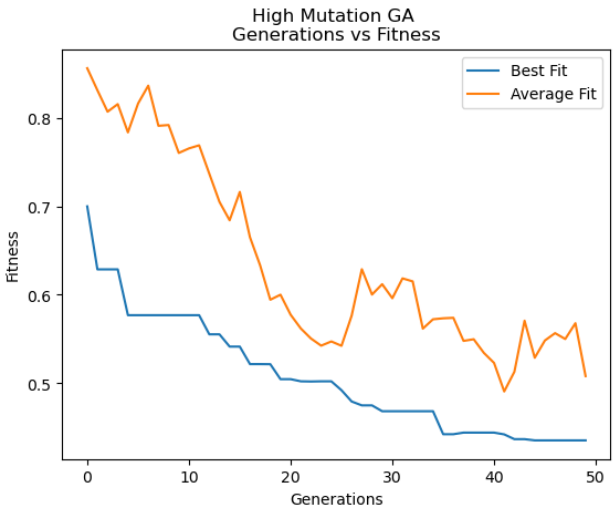


Figure 11: Visual representation of the fitness scores against the number of the generations for both runs of test case 8

Test Case 8 evaluation: The average fitness score for test case 8 was 0.4281. This was a slight reduction (2.21%) from test case 7. This result showcases the increase in mutation had a negligible effect on the output of the genetic algorithm. In Figure 11 the average fitness follows a similar path to that of test case 5, further supporting the claim. The increase in mutation rate increases the amount of random search in our genetic algorithm. Therefore the mutation rate was decreased back to 10% for test case 9.

Since mutation is preventing the genetic algorithms from converging in test case 5, the amount of generations is increased to 100 in test case 9, to check the effect of doubling the generation amount. Due to the average population fitness being relatively similar at generation 50 to the best fitness in test case 5, there should be minimal to no improvement. Due to the doubling of generations, test case 9 is expected to take twice as long as test case 8 to run.

Test Case 9: High Generation GA. The GA has the following traits:

- Generations: 100
- Population: 50
- Text Corpus: 10,000 characters
- Crossover Method: Two Point CrossOver
- Selection Model: Rank Selection
- Mutation Rate: 1 0%
- Mutation Method:Swap, Shuffle, Inverse: 33.33%
- Elitism: 10%
- Stopping Condition: 100 Generations

Results:

High Generation GA details	Run 1	Run 2	Average
CPU Time run $(seconds)$	79.38	79.6	79.49
Best Fitness Score $(FingerDistance\ /Character)$	0.4498	0.418	0.4339

Visualizations:

Run 1 ***Run 2***

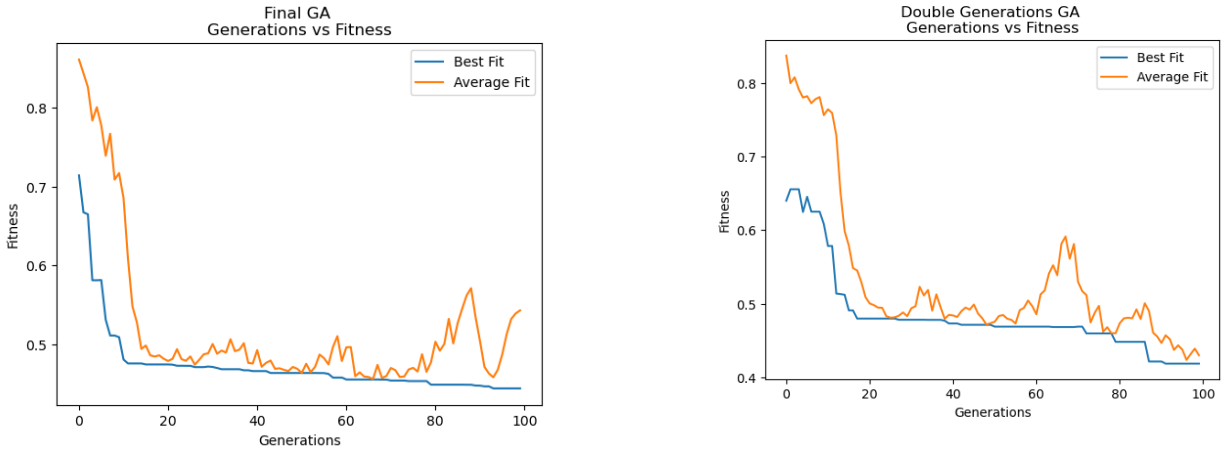


Figure 12: Visual representation of the fitness scores against the number of the generations for both runs of test case 9

Test Case 9 evaluation: The average fitness score for test case 9 is 0.4339. Increasing the number of generations to 100 only had a minimal effect on the score. There was a 5.37% improvement from test case 5. Around double the time of test case 8 was needed to run test case 9 which took an average of 74.49s to run. Doubling the amount of generations had a minimal improvement while doubling the amount of time the GA takes to run.

Figure 13: The best keyboard layout obtained from all the test cases. The best keyboard of test case 7 run 1

Conclusion: The best keyboard obtained from all the test cases was test case 7 first run best keyboard, with a fitness score of 0.4168. It is to note that the best keyboard has all the vowels except the vowel 'I' in the middle row. Moreover the letters "X" , "Z" , ";" and "V" which are some of the least frequently used characters are placed in the worst mathematical position for any letter [5]. The 8 most frequent letters seem to be the "O", "H", "E", "S", "A", "R", "T" and "N" since these are the letters in the starting finger positions.

The best keyboard is a 53.62% improvement over the Azerty layout and a 49.92% improvement over the Qwerty layout. These figures show how much the standard keyboard layouts could be improved. However changing a standard which is well known throughout the world would be close to impossible and not the aim of this research.

To future optimize the GA future testing could be conducted on the GA's parameters. Testing would involve tweaking mutation and elitism parameters of either test case 5 or 7. Different elitism rates were not tested in this study and different elitism rates might drastically improve the genetic algorithm performance. Another possible improvement could be using test case 5 or 7 with single point crossover since the introduction of double point crossover decreased the score of the genetic algorithm. Future testing would also involve using different datasets and languages that do not involve English.

One crucial improvement is to test all the test cases multiple times to eliminate the random effect of the starting population, crossover methods and mutation on the fitness scores. Running each genetic algorithm only twice only gives limited visibility on each test case due to the random probability of genetic algorithms.

The best test case obtained from this study when comparing fitness scores and the time taken to run is test case 6. Test case 6 gives keyboards with the lowest fitness scores and is the most efficient if taking time into consideration.

Statement of completion -

Item	**Completed (Yes/No/Partial)**
Implement ‘base’ genetic algorithm	Yes
Two-point crossover	Yes
Implemented a mutation operation	Yes
Elitism	Yes
Good evaluation and discussion	Yes

References:

[1] C.E.Craddock, “The amulet: A novel by Charles Egbert Craddock”, Project Gutenberg, https://www.gutenberg.org/ebooks/69483 (accessed 01/06/2023).

[2] K.Haviland-Taylor, “A modern trio in an old town by Katharine Haviland-Taylor”, Project Gutenberg, https://www.gutenberg.org/ebooks/69474 (accessed 01/06/2023).

[3] “A Apple Pie and Other Nursery Tales by Unknown”, Project Gutenberg, https://www.gutenberg.org/ebooks/24117 (accessed 01/06/2023).

[4] J.Verne, “Abandoned by Jules Verne” Project Gutenberg, https://www.gutenberg.org/ebooks/33516 (accessed 01/06/2023).

[5] adumb, “Using AI to Create the Perfect Keyboard.” Youtube, https://www.youtube.com/watch?v=EOaPb9wrgDY (accessed 12/23/2022).

[6] T.Fahad, “What is roulette wheel selection?”, Educative, https://www.educative.io/answers/what-is-roulette-wheel-selection (accessed 12/23/2022).

[7] A.Dukka, “Crossover in Genetic Algorithm”, GeeksforGeeks, . https://www.geeksforgeeks.org/crossover-in-genetic-algorithm/ (accessed 12/23/2022).

[8] psil123, “Mutation Algorithms for String Manipulation (GA)”, GeeksforGeeks https://www.geeksforgeeks.org/mutation-algorithms-for-string-manipulation-ga/ (accessed 12/23/2022).