L-Università ta' Malta
Faculty of Information & Communication Technology
Department of Artificial Intelligence

ICS2207 - Machine Learning: Classification, Search and Optimisation

Course Project

Jan Lawrence Formosa
(0435502L)

Statement Of Completion

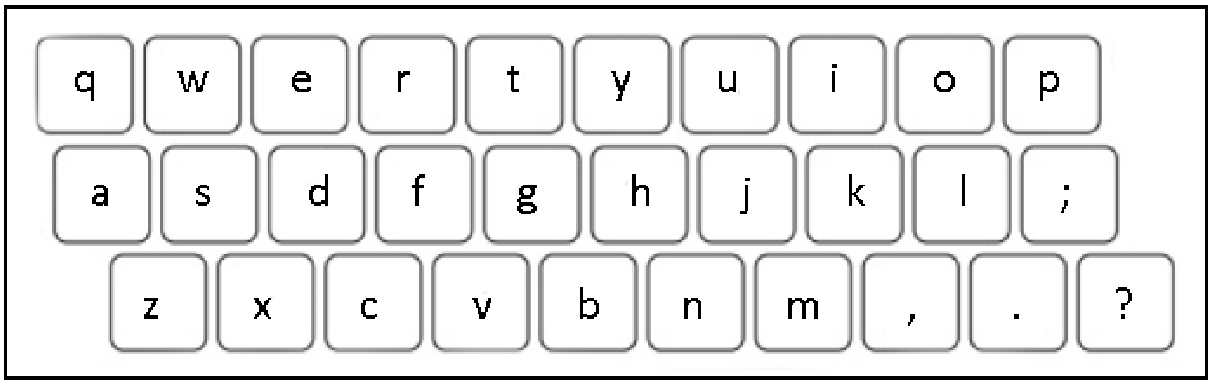
Item	**Completed (Yes/No/Partial)**
Implement ‘base’ genetic algorithm	Yes
Two-point crossover	Yes
Implemented a mutation operation	Yes
Elitism	Yes
Good evaluation and discussion	Yes

Introduction

The original QWERTY keyboard layout we are all familiar with today was created all the way back in 1874 by Christopher Latham Scholes.

In the 149 years since its creation, the technology we use on our devices has taken leaps and bounds however the keyboard layout has remained unchanged.

This reluctance to change to a better layout is likely due to the fact that people simply got used to the QWERTY layout, inefficient and unintuitive as it may be.



Traditional QWERTY Keyboard Layout

In spite of this reluctance to change, the fact remains that more optimal keyboard layouts do exist.

The scope of this project is to generate new keyboard layouts via Genetic Algorithms with the goal of trying to discover the most optimal keyboard layout possible.

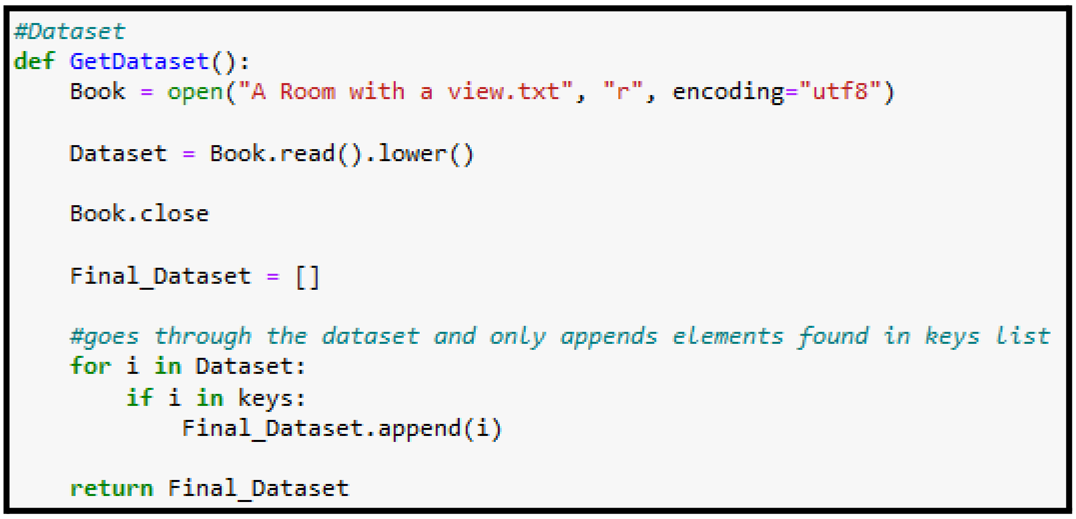
This idea was already explored in the YouTube video[1] provided in the project description, however we will be building on it and adding additional features to the Genetic Algorithms.

Design Decisions

Choice Of Corpus

The book “A Room with a View”[2] was downloaded from the Gutenberg website[3] to be used as the corpus to test the efficiency of the keyboard layouts.

After downloading the book as plain text into a text file, the program filters the characters from the book and appends each valid character to a list. This is done to ensure that only accepted characters end up in our dataset.



```
#Dataset
def GetDataset():
    Book = open("A Room with a view.txt", "r", encoding="utf8")

    Dataset = Book.read().lower()

    Book.close

    Final_Dataset = []

    #goes through the dataset and only appends elements found in keys list
    for i in Dataset:
        if i in keys:
            Final_Dataset.append(i)

    return Final_Dataset
```

Code responsible for this procedure

These accepted characters will be discussed further underneath the Chromosome Design sub section.

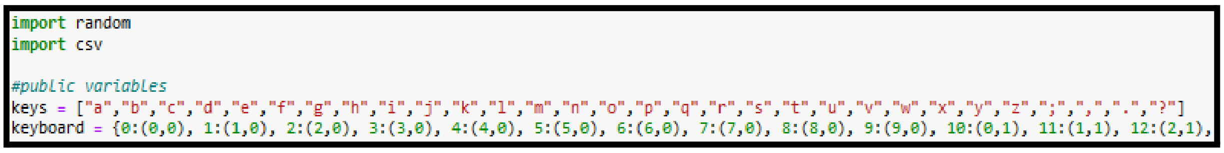
Chromosome Design

Each chromosome of the Genetic Algorithm contains a keyboard layout and each chromosome makes up one element in the population. Therefore the larger the population size, the more chromosomes per generation.

The characters our keyboards will use are defined in the “keys” list and include all the letters of the alphabet plus four punctuation marks. As mentioned before, the dataset is pruned to only include the elements in this list.

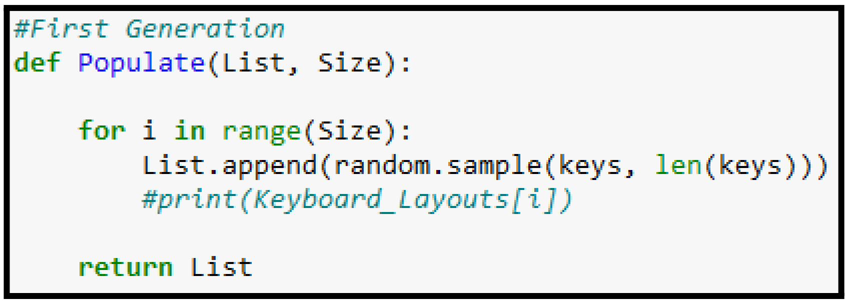
At runtime, the “Populate” function is called. Based on the population size provided in our main class, this function will fill each chromosome with a randomly generated keyboard layout.

Each character is also assigned a coordinate, relative to said character's position in the list.



```python
import random
import csv

#public variables
keys = ["a","b","c","d","e","f","g","h","i","j","k","l","m","n","o","p","q","r","s","t","u","v","w","x","y","z",";",",",".","?"]
keyboard = {0:(0,0), 1:(1,0), 2:(2,0), 3:(3,0), 4:(4,0), 5:(5,0), 6:(6,0), 7:(7,0), 8:(8,0), 9:(9,0), 10:(0,1), 11:(1,1), 12:(2,1),
```

Code which defines the valid characters as well as their coordinates



```python
#First Generation
def Populate(List, Size):

    for i in range(Size):
        List.append(random.sample(keys, len(keys)))
        #print(Keyboard_Layouts[i])

    return List
```

Generating random keyboard layouts for the first generation of the Genetic Algorithm

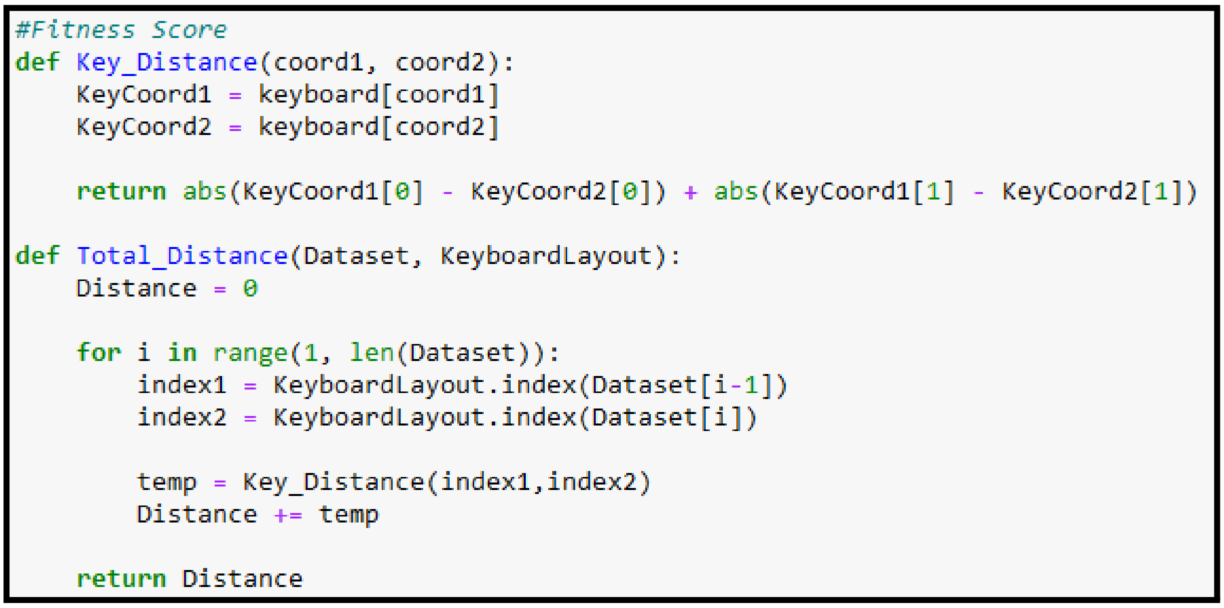
The coordinates assigned to the characters are used to measure the distance between two keys and ultimately, the fitness score of the keyboard layout. This is done using the “Total_Distance” method.

In the “Total_Distance” method, the aforementioned corpus is passed along with a keyboard layout. The position of the first and second characters of the corpus, relative to the keyboard layout, are stored into two variables, namely “index1” and “index2”. This means that if the first character in the corpus is “e” and the character “e” is in position 10 in the keyboard layout, then the number 10 is stored into the variable. These variables are then passed into another method called “Key_Distance”.

“Key_Distance” calculates a numerical representation for the distance between two keys. This is done by using the index variables to obtain the coordinates for the two characters and finding the distance between the two coordinates.

Let us assume “index1” contained the number “10” and “index2” contained the number “15”. By making use of the coordinates list called “keyboard”, which is defined under public variables, “KeyCoord1” will contain the coordinates (0,1) whilst “KeyCoord2” will contain the coordinates (5,1)). Based on these coordinates the distance between the two keys is found to be 5.

This procedure is repeated for every character in the corpus and once complete, all the distances are tallied and a total distance, or Fitness Score, is returned.



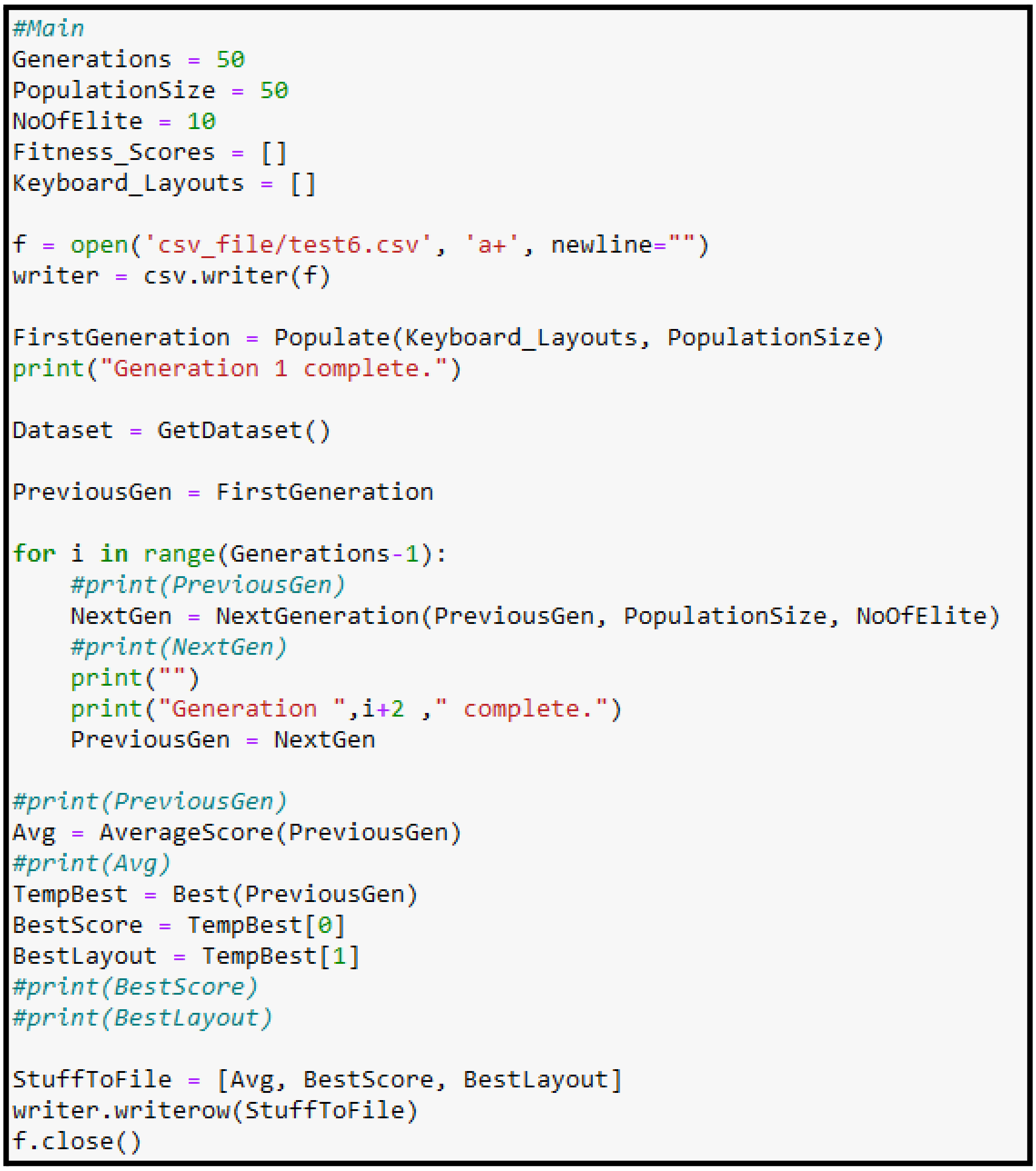
```python
#Fitness Score
def Key_Distance(coord1, coord2):
    KeyCoord1 = keyboard[coord1]
    KeyCoord2 = keyboard[coord2]

    return abs(KeyCoord1[0] - KeyCoord2[0]) + abs(KeyCoord1[1] - KeyCoord2[1])

def Total_Distance(Dataset, KeyboardLayout):
    Distance = 0

    for i in range(1, len(Dataset)):
        index1 = KeyboardLayout.index(Dataset[i-1])
        index2 = KeyboardLayout.index(Dataset[i])

        temp = Key_Distance(index1,index2)
        Distance += temp

    return Distance
```

The aforementioned functions which calculate the fitness score of a keyboard layout

Once the fitness score of every single keyboard layout is calculated, a number of operations take place based on the parameters decided in the “main” class.

Through these operations, the next generation of the Genetic Algorithm is created while also storing the results of the average Fitness Score, Best Fitness Score and Best Keyboard Layout of each generation in a .csv file.

These operations will be discussed at length below.



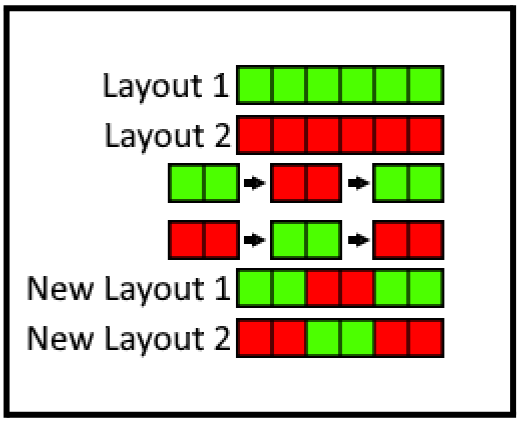
```
#Main
Generations = 50
PopulationSize = 50
NoOfElite = 10
Fitness_Scores = []
Keyboard_Layouts = []

f = open('csv_file/test6.csv', 'a+', newline="")
writer = csv.writer(f)

FirstGeneration = Populate(Keyboard_Layouts, PopulationSize)
print("Generation 1 complete.")

Dataset = GetDataset()

PreviousGen = FirstGeneration

for i in range(Generations-1):
    #print(PreviousGen)
    NextGen = NextGeneration(PreviousGen, PopulationSize, NoOfElite)
    #print(NextGen)
    print("")
    print("Generation ",i+2 ," complete.")
    PreviousGen = NextGen

#print(PreviousGen)
Avg = AverageScore(PreviousGen)
#print(Avg)
TempBest = Best(PreviousGen)
BestScore = TempBest[0]
BestLayout = TempBest[1]
#print(BestScore)
#print(BestLayout)

StuffToFile = [Avg, BestScore, BestLayout]
writer.writerow(StuffToFile)
f.close()
```

“main” class responsible for test case parameters calling different functions

Crossover Design

A Two Point Crossover function was designed to generate every subsequent generation in the Genetic Algorithm after the first one.

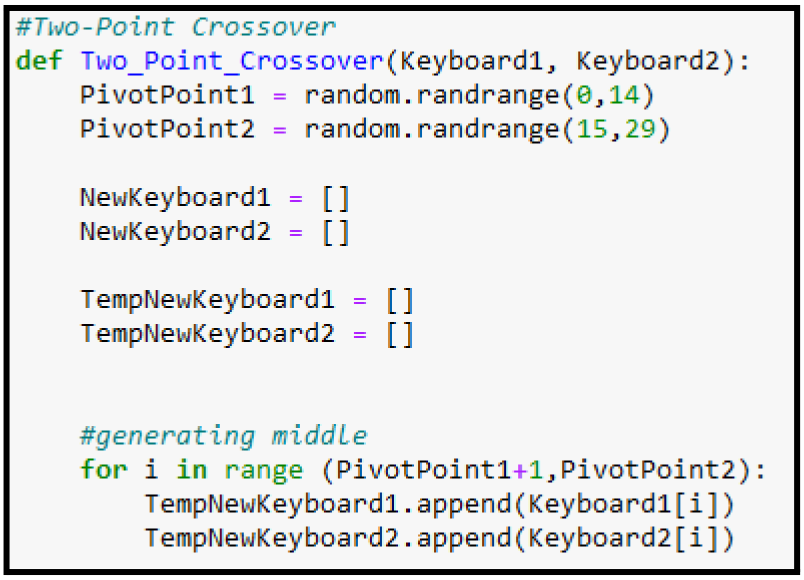
The Two Point Crossover works by taking two keyboard layouts and breaking them in two random different spots. This gives us three small parts of each layout: a front, a middle and an end. The middle of each layout is swapped and then all the pieces are fused together to create two new keyboard layouts.



Simple diagram depicting the logic in Two Point Crossover

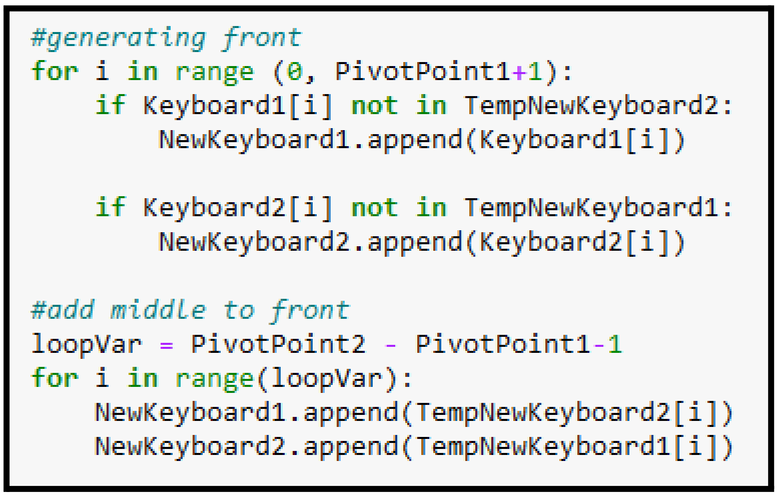
To start, two unique random numbers depicting two points on the list are generated and stored in Pivot Point variables. Temporary lists are then used to store the characters in between these two points. These temporary lists represent the 'middle' of the list.

Pivot Points and 'middle' are created



```
#Two-Point Crossover
def Two_Point_Crossover(Keyboard1, Keyboard2):
    PivotPoint1 = random.randrange(0,14)
    PivotPoint2 = random.randrange(15,29)

    NewKeyboard1 = []
    NewKeyboard2 = []

    TempNewKeyboard1 = []
    TempNewKeyboard2 = []


    #generating middle
    for i in range (PivotPoint1+1,PivotPoint2):
        TempNewKeyboard1.append(Keyboard1[i])
        TempNewKeyboard2.append(Keyboard2[i])
```

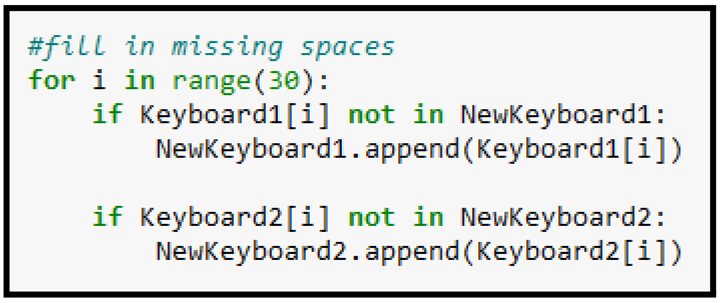
The characters in the 'front' of both layouts are appended to empty lists which will contain the new layouts while a check also takes place to make sure any repeated characters with the new middle are removed. The swapped 'middles' are then appended to these lists.



```
#generating front
for i in range (0, PivotPoint1+1):
    if Keyboard1[i] not in TempNewKeyboard2:
        NewKeyboard1.append(Keyboard1[i])

    if Keyboard2[i] not in TempNewKeyboard1:
        NewKeyboard2.append(Keyboard2[i])

#add middle to front
loopVar = PivotPoint2 - PivotPoint1-1
for i in range(loopVar):
    NewKeyboard1.append(TempNewKeyboard2[i])
    NewKeyboard2.append(TempNewKeyboard1[i])
```

'front' is created and swapped 'middle' is appended

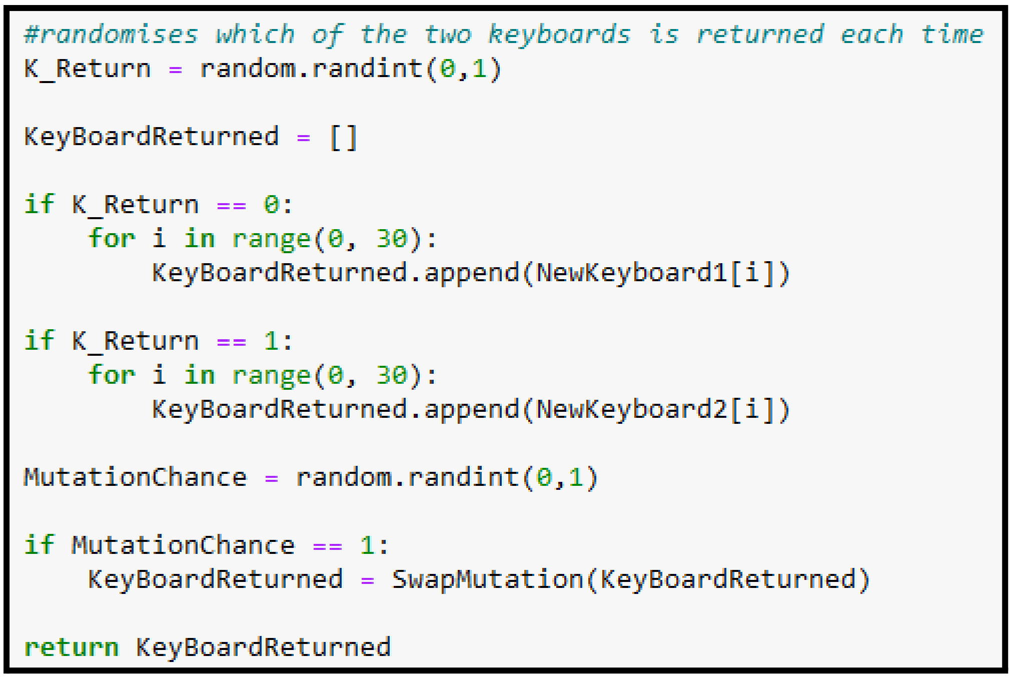
Each of the original layouts is then traversed in order and any element not found in the new layouts are appended to the back of them. This results in a complete list which is not missing any of the repeated characters removed earlier.



```
#fill in missing spaces
for i in range(30):
    if Keyboard1[i] not in NewKeyboard1:
        NewKeyboard1.append(Keyboard1[i])

    if Keyboard2[i] not in NewKeyboard2:
        NewKeyboard2.append(Keyboard2[i])
```

Empty spaces are filled in

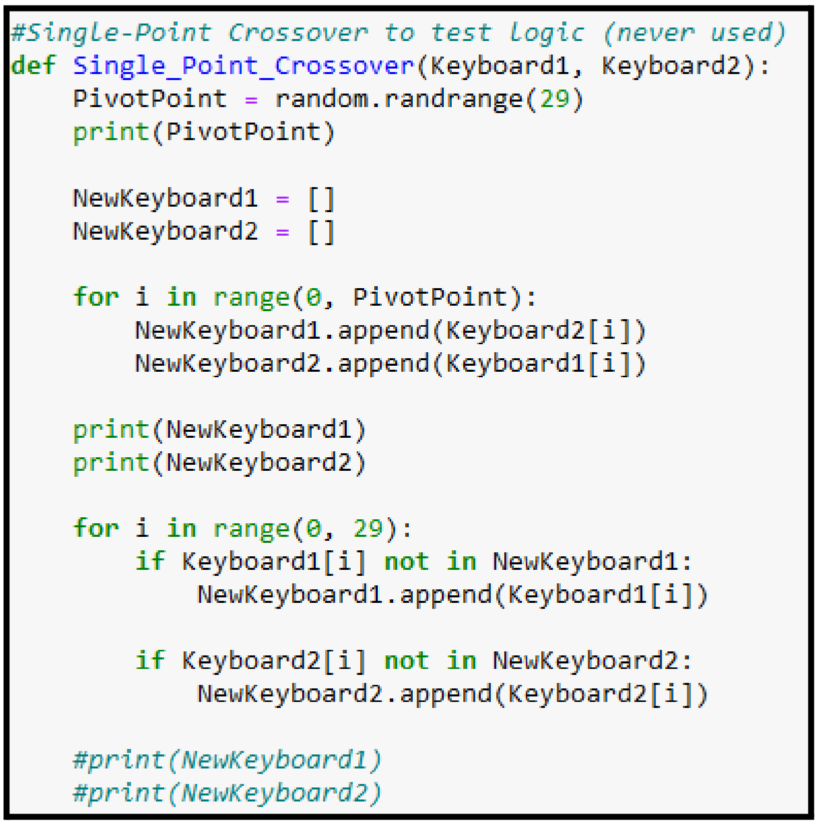
Only one of the two new layouts is returned and added as a new chromosome in the next generation. To decide which of these layouts is selected, a simple boolean check occurs. A similar check is also conducted to check if the keyboard should also be mutated before it is returned.



```
#randomises which of the two keyboards is returned each time
K_Return = random.randint(0,1)

KeyBoardReturned = []

if K_Return == 0:
    for i in range(0, 30):
        KeyBoardReturned.append(NewKeyboard1[i])

if K_Return == 1:
    for i in range(0, 30):
        KeyBoardReturned.append(NewKeyboard2[i])

MutationChance = random.randint(0,1)

if MutationChance == 1:
    KeyBoardReturned = SwapMutation(KeyBoardReturned)

return KeyBoardReturned
```

Deciding which new layout to return and if it should be mutated before

A Single Point Crossover function was also designed to gain a better understanding of crossover operations before attempting Two Point Crossover but it is never used.



```
#Single-Point Crossover to test logic (never used)
def Single_Point_Crossover(Keyboard1, Keyboard2):
    PivotPoint = random.randrange(29)
    print(PivotPoint)

    NewKeyboard1 = []
    NewKeyboard2 = []

    for i in range(0, PivotPoint):
        NewKeyboard1.append(Keyboard2[i])
        NewKeyboard2.append(Keyboard1[i])

    print(NewKeyboard1)
    print(NewKeyboard2)

    for i in range(0, 29):
        if Keyboard1[i] not in NewKeyboard1:
            NewKeyboard1.append(Keyboard1[i])

        if Keyboard2[i] not in NewKeyboard2:
            NewKeyboard2.append(Keyboard2[i])

    #print(NewKeyboard1)
    #print(NewKeyboard2)
```

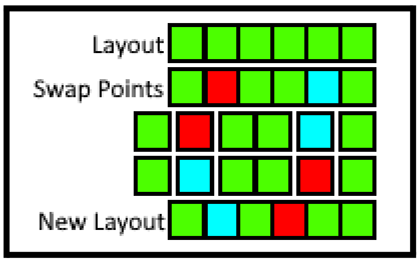
Unused Single Point Crossover function

Mutation Operation

If the boolean check in “Two_Point_Crossover” rolls a “1” then the “SwapMutation” function is called.

As the name implies, the mutation taking place is a swap mutation.

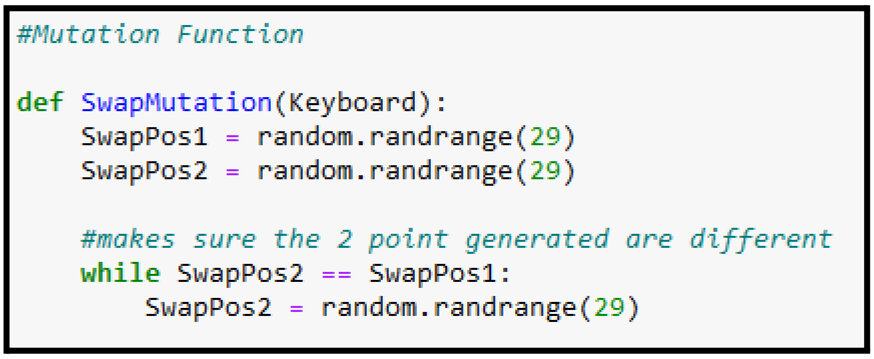
The swap mutation consists of randomly selecting two points on a list and swapping the elements at those points.



Swap Mutation Logic

This was implemented in the program in a few simple steps.

First two unique random numbers are generated and stored in variables.



```
#Mutation Function

def SwapMutation(Keyboard):
    SwapPos1 = random.randrange(29)
    SwapPos2 = random.randrange(29)

    #makes sure the 2 point generated are different
    while SwapPos2 == SwapPos1:
        SwapPos2 = random.randrange(29)
```

The order to following operations are partially reversed based on which of the two random numbers is larger.

Assuming the second number is larger, everything up to the first point’s index is appended to an empty list. Then the element at the second point of the original layout is added.

```
if SwapPos2 > SwapPos1:
    for i in range(0, SwapPos1):
        NewKeyboard.append(Keyboard[i])

    NewKeyboard.append(Keyboard[SwapPos2])
```

This is followed by appending all the elements up to the second point's index.

```
for i in range(SwapPos1+1, SwapPos2):
    NewKeyboard.append(Keyboard[i])
```

After that is done, the element at the first point in the original list is appended followed by all the remaining elements.

```
    NewKeyboard.append(Keyboard[SwapPos1])

    for i in range(SwapPos2, 29):
        NewKeyboard.append(Keyboard[i+1])

return NewKeyboard
```

Had the first number been larger then the same process would have occurred but with every instance of the SwapPoint variables swapped.

```
if SwapPos1 > SwapPos2:
    for i in range(0, SwapPos2):
        NewKeyboard.append(Keyboard[i])

    NewKeyboard.append(Keyboard[SwapPos1])

    for i in range(SwapPos2+1, SwapPos1):
        NewKeyboard.append(Keyboard[i])

    NewKeyboard.append(Keyboard[SwapPos2])

    for i in range(SwapPos1, 29):
        NewKeyboard.append(Keyboard[i+1])
```

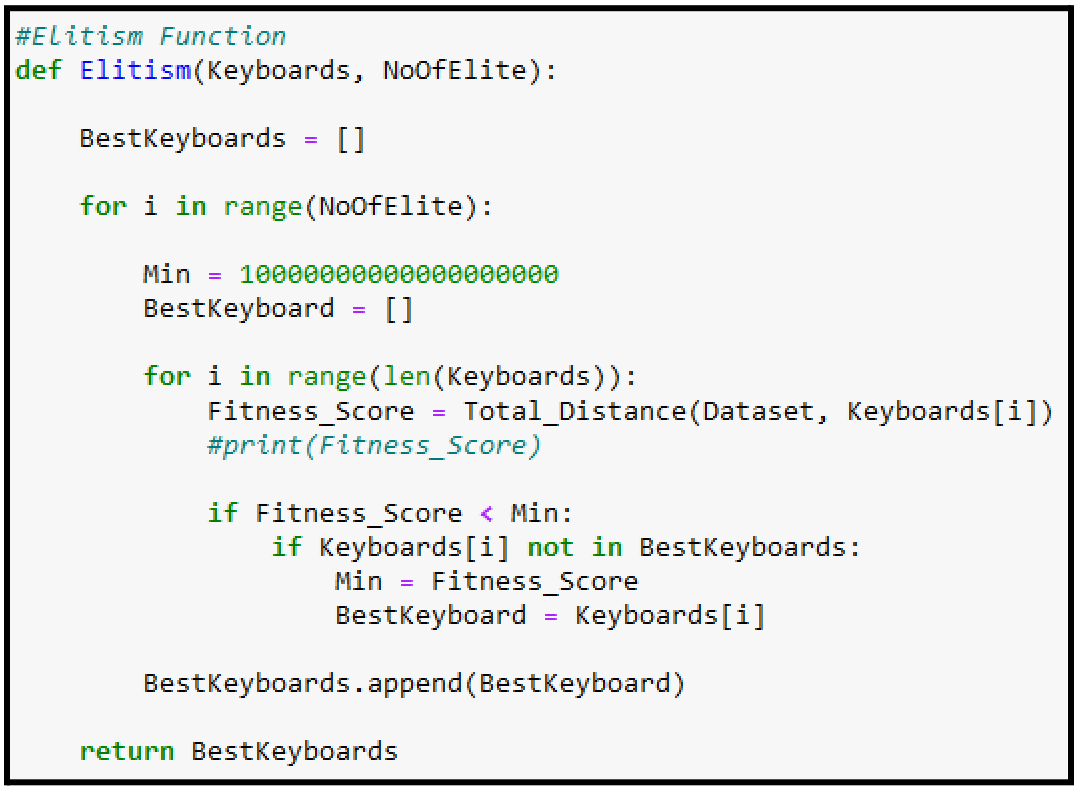
Elitism

The elitism operation is responsible for saving the best keyboards of each generation and adding them to the next generation of the Genetic Algorithm.

In the operation a simple loop occurs where every layout in a generation is compared to the current lowest score of the layouts tested up to that point and if it is lower than everything else then it takes over the “BestKeyboard” variable.

This loop tests every single layout in the generation. Once complete, the current layout occupying the “BestKeyboard” variable is the layout with the lowest score and is appended to the “BestKeyboards” list.

Once a layout has been saved to “BestKeyboards” it is skipped on subsequent checks so that the next best keyboard can be added to the list. The amount of layouts to be carried over into the next generation is dictated in the “main” class.

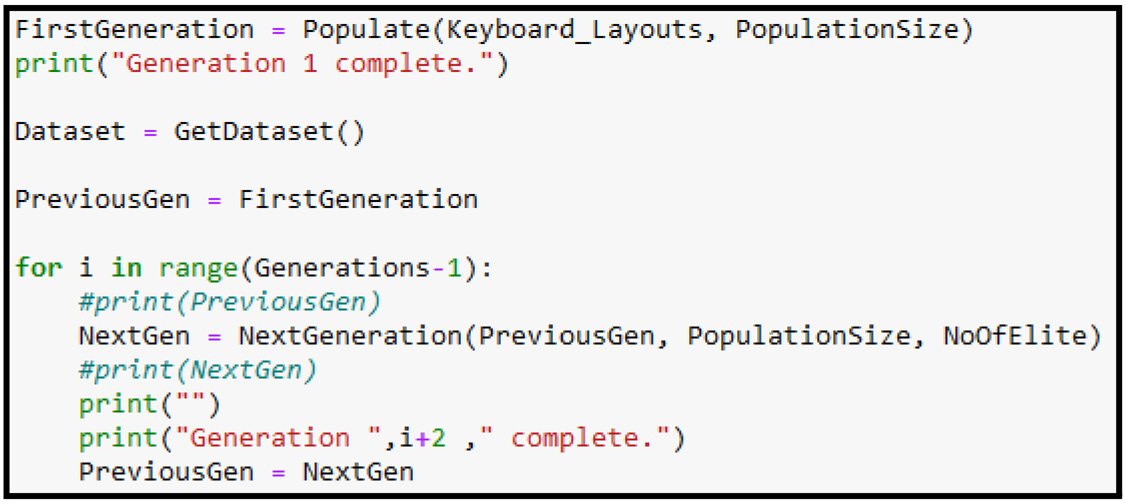


```python
#Elitism Function
def Elitism(Keyboards, NoOfElite):

    BestKeyboards = []

    for i in range(NoOfElite):

        Min = 10000000000000000000
        BestKeyboard = []

        for i in range(len(Keyboards)):
            Fitness_Score = Total_Distance(Dataset, Keyboards[i])
            #print(Fitness_Score)

            if Fitness_Score < Min:
                if Keyboards[i] not in BestKeyboards:
                    Min = Fitness_Score
                    BestKeyboard = Keyboards[i]

        BestKeyboards.append(BestKeyboard)

    return BestKeyboards
```

The elitism function

<u>Chromosome Design (Continued)</u>

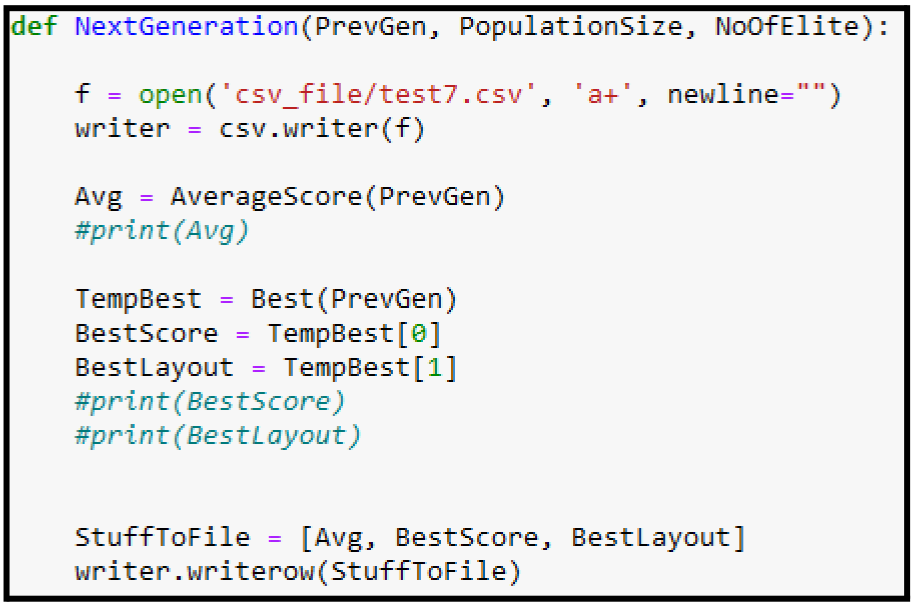
Now that the operations have been explained, it is easier to understand what is happening when a new generation is being created.



```python
FirstGeneration = Populate(Keyboard_Layouts, PopulationSize)
print("Generation 1 complete.")

Dataset = GetDataset()

PreviousGen = FirstGeneration

for i in range(Generations-1):
    #print(PreviousGen)
    NextGen = NextGeneration(PreviousGen, PopulationSize, NoOfElite)
    #print(NextGen)
    print("")
    print("Generation ",i+2 ," complete.")
    PreviousGen = NextGen
```

As can be seen in the “main” class, the first population is generated and passed in the “NextGeneration” function. This function loops according to how many generations were specified before. Every “NextGen” list then becomes the “PreviousGen” in the next iteration of the loop keeping the cycle going.

Before any changes happen inside the “NextGeneration” function, the average fitness score ,best fitness score and best layout are calculated and saved to a file.



```python
def NextGeneration(PrevGen, PopulationSize, NoOfElite):

    f = open('csv_file/test7.csv', 'a+', newline="")
    writer = csv.writer(f)

    Avg = AverageScore(PrevGen)
    #print(Avg)

    TempBest = Best(PrevGen)
    BestScore = TempBest[0]
    BestLayout = TempBest[1]
    #print(BestScore)
    #print(BestLayout)


    StuffToFile = [Avg, BestScore, BestLayout]
    writer.writerow(StuffToFile)
```

```
#Average and Best Score Functions
def AverageScore(Keyboard):
    Fitness_Scores = []
    TotalScore = 0
    for i in range(len(Keyboard)):
        Fitness_Scores.append(Total_Distance(Dataset, Keyboard[i]))
        temp = Total_Distance(Dataset, Keyboard[i])
        TotalScore += temp

        #print(Fitness_Scores)
        #print(TotalScore)

    Average = TotalScore / (len(Keyboard))
    #print(Average)

    return Average

def Best(Keyboard):
    BestScore = 100000000000000
    BestKeyboard = []

    for i in range(len(Keyboard)):
            Fitness_Score = Total_Distance(Dataset, Keyboard[i])
            #print(Fitness_Score)

            if Fitness_Score < BestScore:
                BestKeyboard = Keyboard[i]
                BestScore = Fitness_Score

    #print(BestScore)

    return [BestScore, BestKeyboard]
```

Simple functions to calculate the average score, best score and best layout

After writing the values to a file, the elitism function is called to get the best layouts of the previous generation. These are then the first layouts added to the next generation in order to ensure that the best layouts always carry over across generations.

```
ElitsmKey = Elitism(PrevGen, NoOfElite)
#print(ElitsmKey)

Keyboard = ElitsmKey
```

For the remaining spaces in the population, two random layouts from the previous generation will be passed into the “Two_Point_Crossover” function and the returned layout will be added to the new generation.

```
for i in range(PopulationSize - len(Keyboard)):
    #gets two random indexes to use for TCP
    TPC_index1 = random.randint(0, (len(PrevGen)-1))
    TPC_index2 = random.randint(0, (len(PrevGen)-1))

    #makes sure the 2 generated indexes are unique
    while TPC_index1 == TPC_index2:
        TPC_index2 = random.randint(0, len(PrevGen)-1)

    #performs Two Point Crossover on 2 of the layouts randomly selected from the generation
    Keyboard.append(Two_Point_Crossover(PrevGen[TPC_index1], PrevGen[TPC_index2]))

return Keyboard
```

Once the new generation has been created, it will be returned to “main” where the cycle can continue until the loop is complete.

Note that the “AverageScore” and “Best” functions as well as the code to save their results to a file are also present in “main” to ensure that all the results are saved since the loop starts from a “+1” increment.

```
f = open('csv_file/test7.csv', 'a+', newline="")
writer = csv.writer(f)
```

```
Avg = AverageScore(PreviousGen)
#print(Avg)
TempBest = Best(PreviousGen)
BestScore = TempBest[0]
BestLayout = TempBest[1]
#print(BestScore)
#print(BestLayout)

StuffToFile = [Avg, BestScore, BestLayout]
writer.writerow(StuffToFile)
f.close()
```

Evaluation

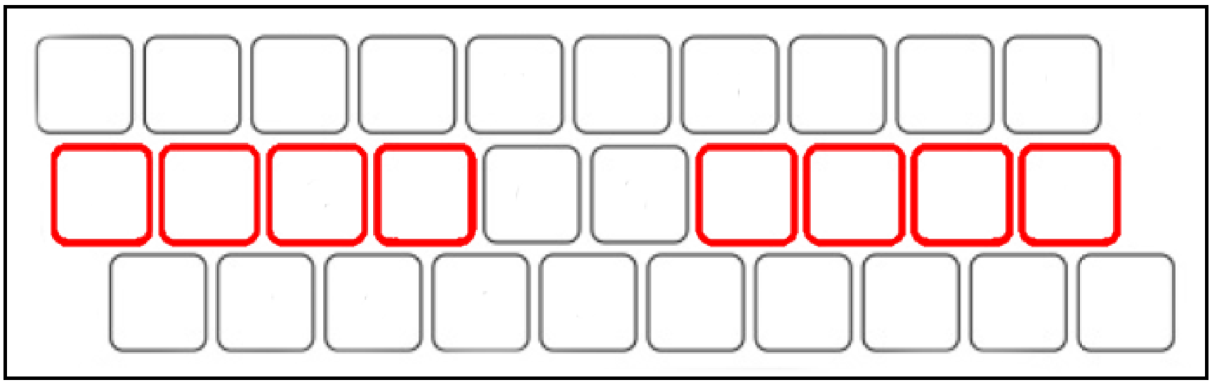
In order to get a broad sample of data to make deductions on, eight unique test cases were run.

These test cases varied in population size, generation size, number of keyboards to carry over generations using elitism and whether or not mutation would ever occur.

The average score and best score will be plotted as a line chart across all generations to make it easier to visualise trends and things of note as the Genetic Algorithms progressed.

A visualisation of what the best layout would look like on a standard keyboard will also be shown.

Assuming a QWERTY layout we know that the fingers will be resting at the “a”,”s”,”d”,”f” and “j”,”k”,”l”,”;” key positions which means the the following positions would be the new starting positions:

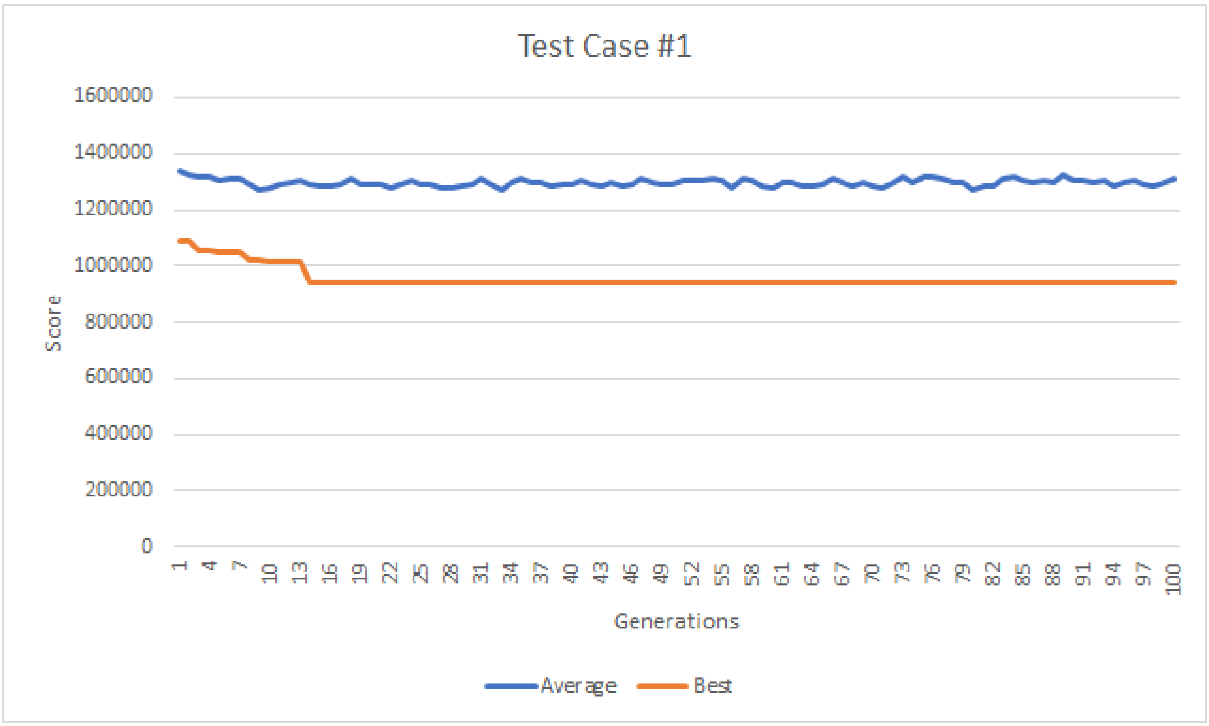


It should also be noted that the closer an element is to the centre of the layout, the more important it is. With reference to this specific corpus the most used characters, that is, the most important, will naturally be in the middle to minimise the fitness score.

The image of the best scoring layout is generated using the Print_Keyboard Jupyter Notebook file.

Test Case #1

Population Size: 100
Number of Generations: 100
Keyboards to carry over via Elitism: 5
Mutation: Yes



This was the first Genetic Algorithm run so there was nothing to compare it to.

The elitism is functioning correctly as it occasionally decreases when a new more optimal layout is found but never increases.

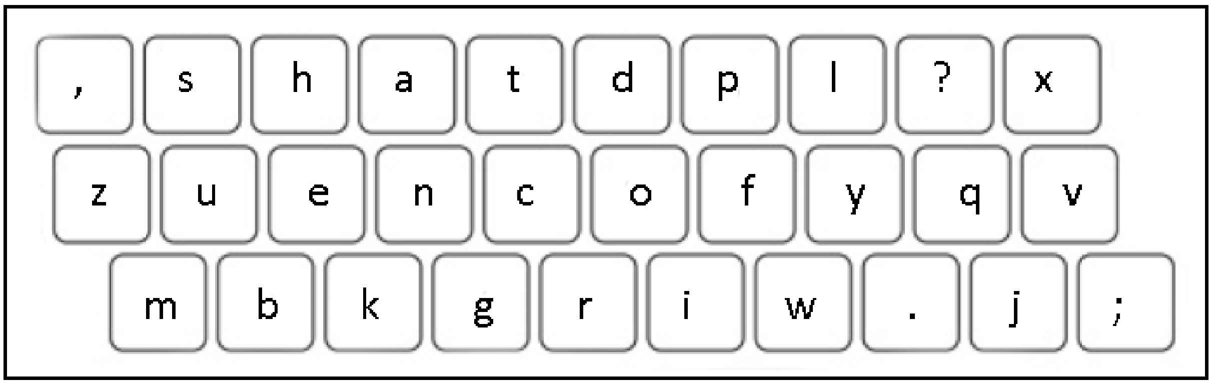
While the average quickly plummets initially, it spends the next 90 generations spiking up and down. The averages would constantly fluctuate between 1,270,000 and 1,330,000 after the first generation.

Generation 9 had an average score of 1,270,231 while the Generation 100 had an average score of 1,311,643.

This highly volatile average is most likely due to only 5% of the population being elite layouts.

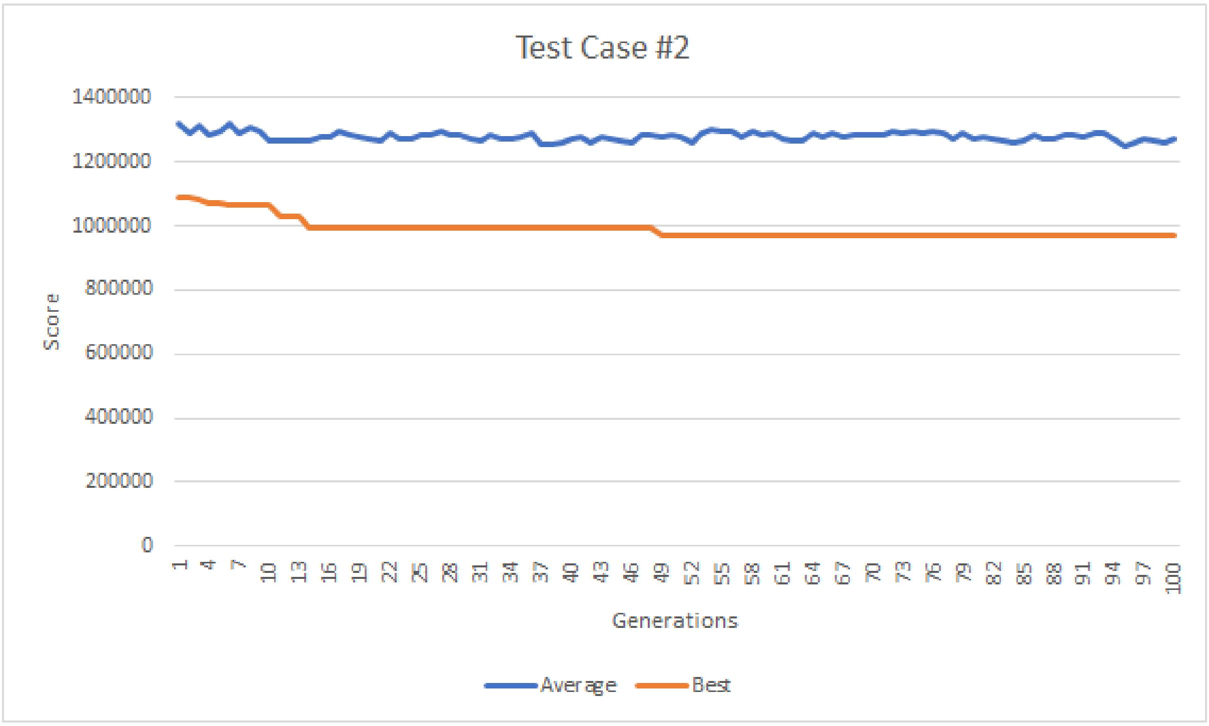
Having so few elite layouts means that the average score of the generation was still mainly determined by the random 95% of layouts and the average score never consistently improved past the very first generation.

The best layout generated by this Genetic Algorithm with a score of 941,963 is the following:



<u>Test 2</u>
Population Size: 100
Number of Generations: 100
Keyboards to carry over via Elitism: 10
Mutation: Yes

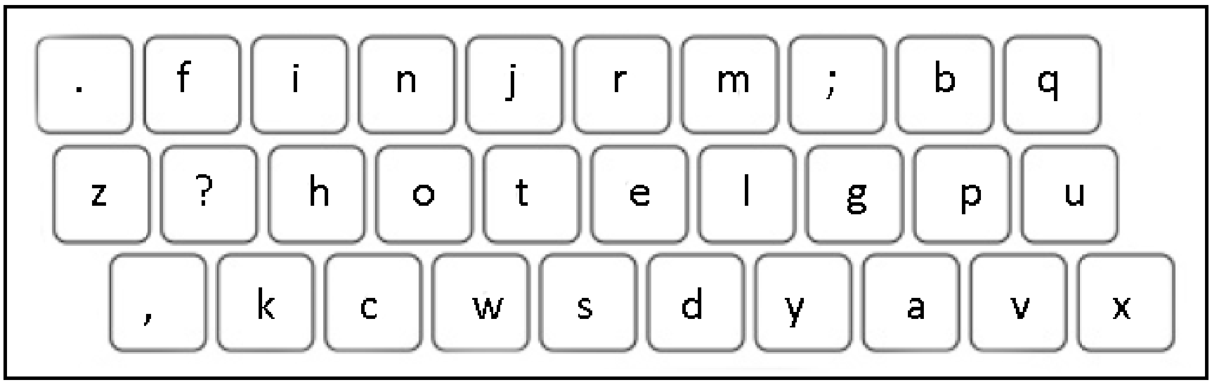
Test Case #2 differs from the first one as the number of elite layouts are double. With this change a lower average is expected to appear much more consistently as the probability of choosing an elite layout has doubled.



As expected, adding an extra 5 elite layouts, thus increasing the number of elite layouts to 10% of the population proved to help lower the average across the generations.

The average score only ever increased to over 1,300,000 once again after the 8th generation. This is a big difference from the previous Genetic Algorithm where the average score would constantly shoot up and down.

The best layout generated by this Genetic Algorithm with a score of 973,913 is the following:

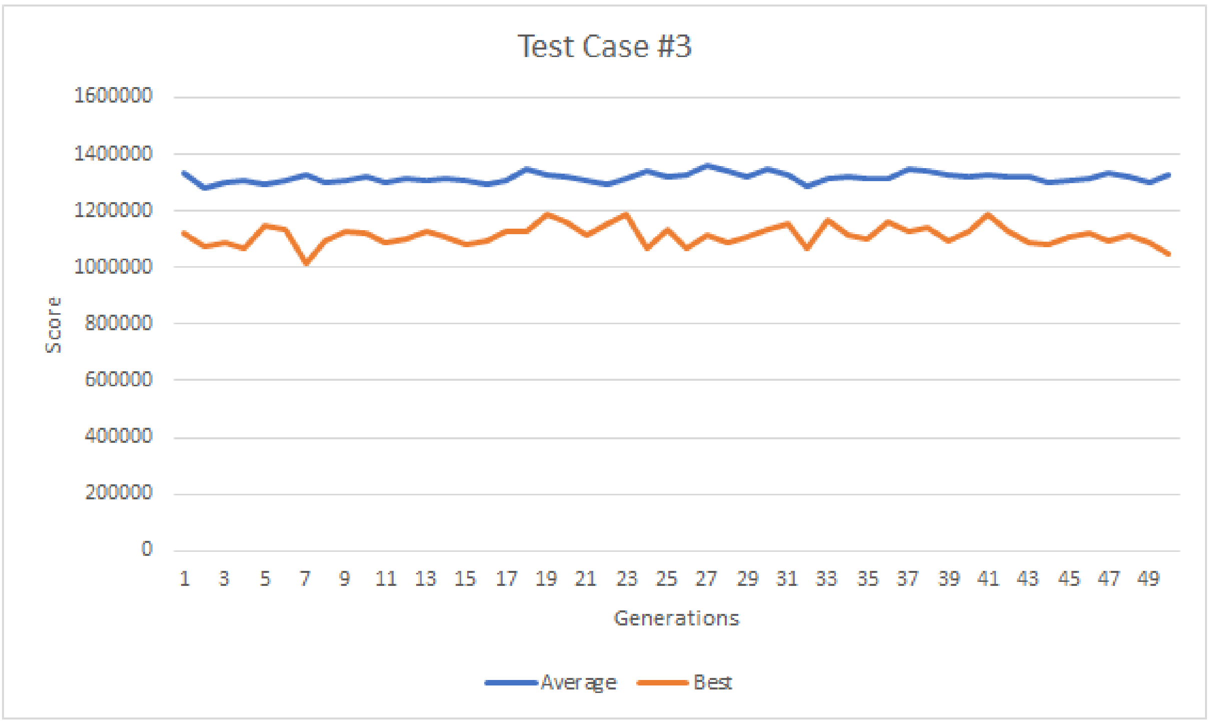


Test Case #3

Population Size: 50
Number of Generations: 50
Keyboards to carry over via Elitism: 0
Mutation: Yes

The population size and generation size for the remaining test cases have been halved.

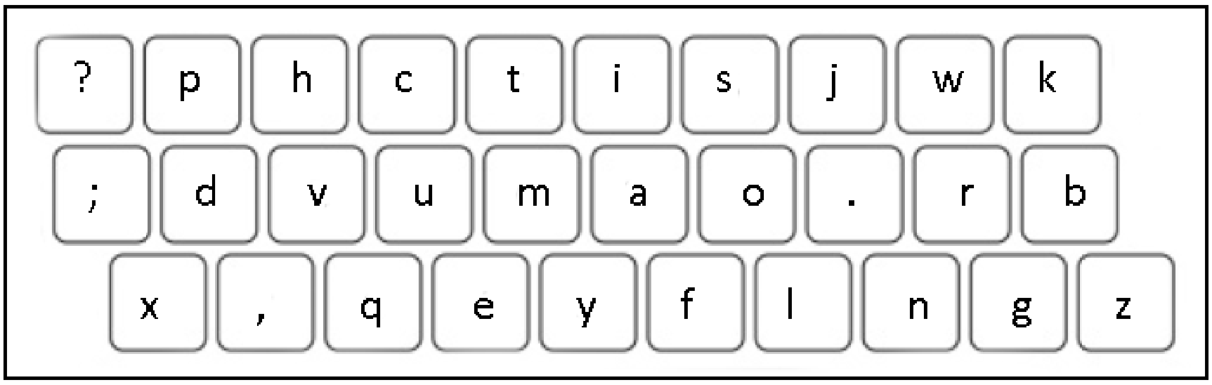
In this test case there are 0 elite layouts which means the best score will change with each generation and there is no gradual improvement in the scores as every generation the entire population is completely different.



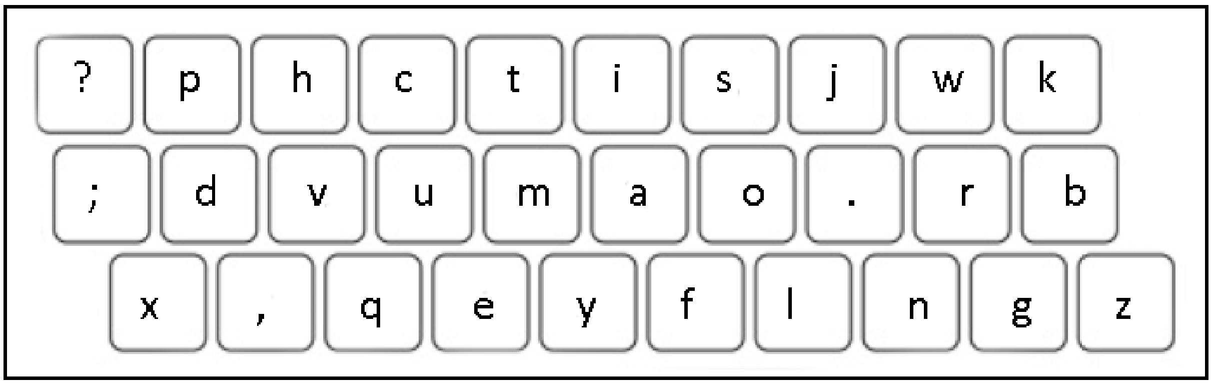
As can be seen above, the best layout is different every generation. In the event of even one elite layout this would never happen as the best score would be stored.

It can also be seen that some average scores in later generations are higher than the average score in the first generation, something that has never happened before because of Elitism.

As the best layout is different every time, our Genetic Algorithm ends believing the best layout generated with a score of 1,045,884 is the following:



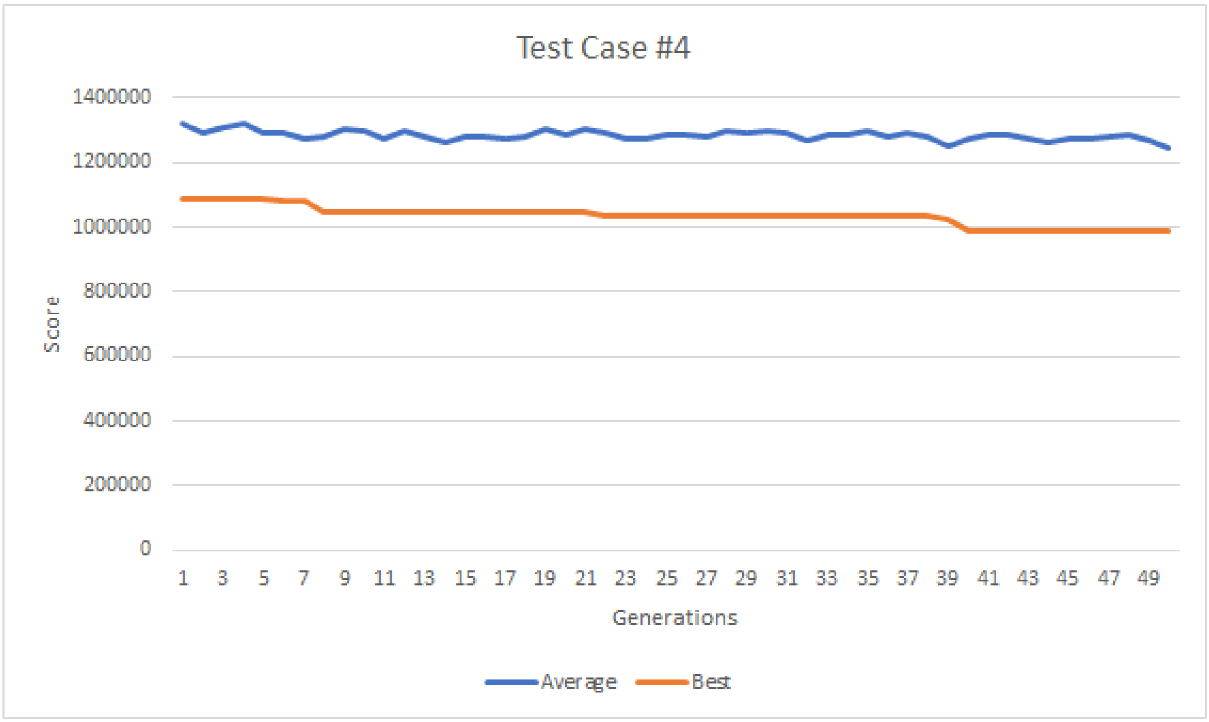
If there was elitism it would be noted that the best layout, with a score of 1,013,772, is actually the following:



<u>Test Case #4</u>

Population Size: 50
Number of Generations: 50
Keyboards to carry over via Elitism: 5
Mutation: Yes

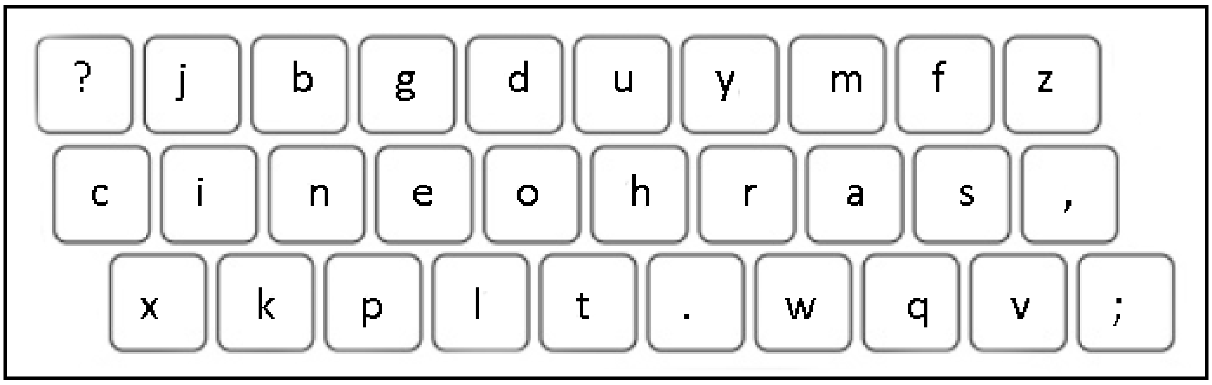
This test case is very similar to Test Case #1 however the population size and amount of generations have been halved.



Similarly to Test Case #1 the averages still fluctuated a lot however this time in a lower score range. The average scores constantly fluctuate between 1,250,00 - 1,300,000.

Other than that the results were very similar to what was observed in Test Case #1.

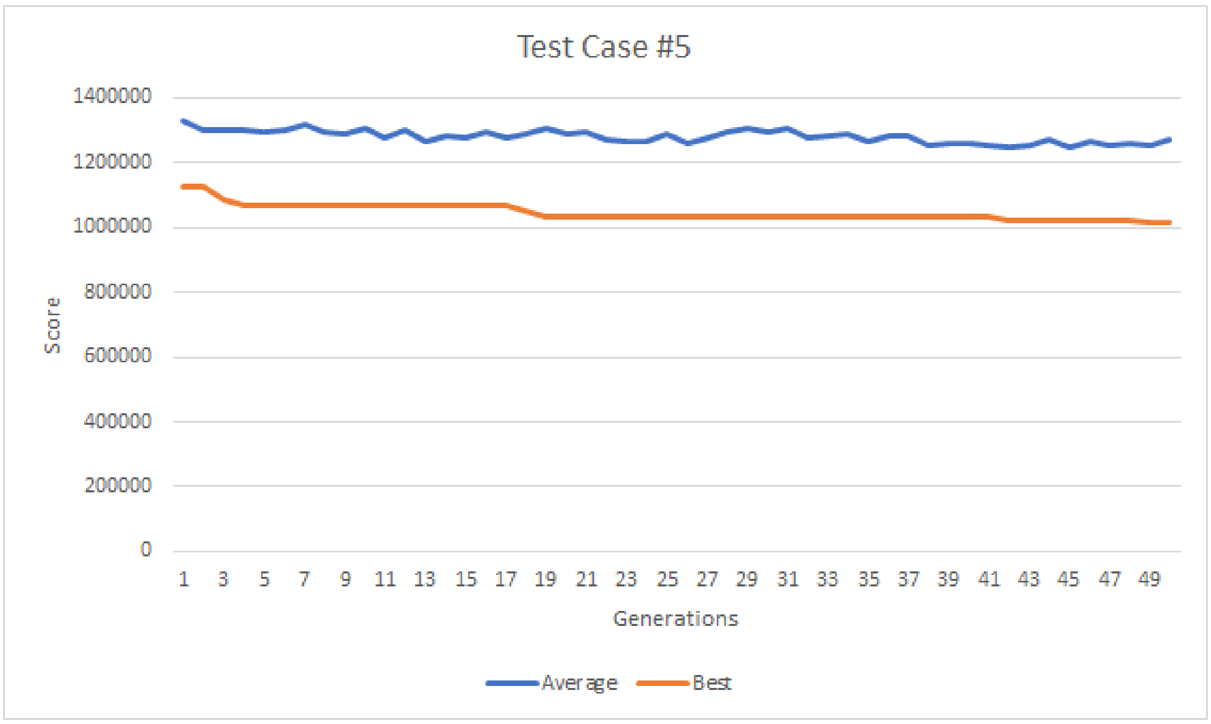
The best layout generated by this Genetic Algorithm with a score of 989,801 is the following:



Test Case #5

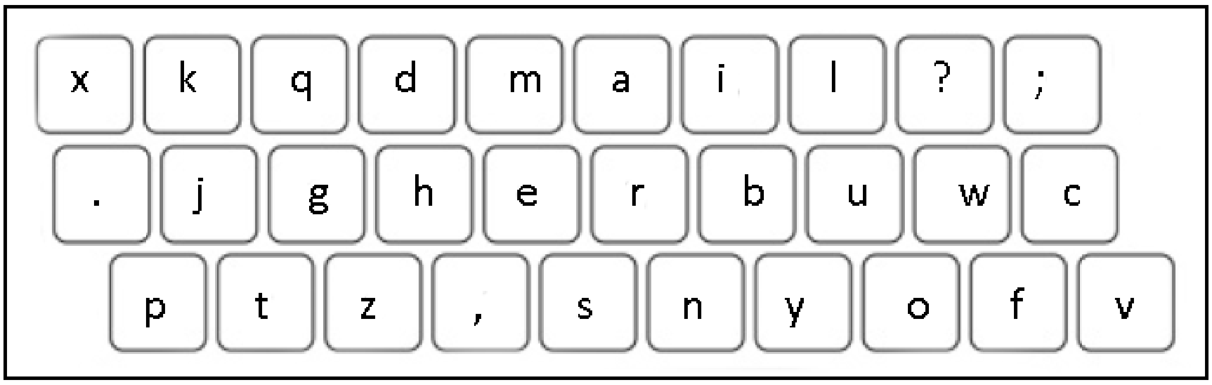
Population Size: 50
Number of Generations: 50
Keyboards to carry over via Elitism: 5
Mutation: No

This test case is almost identical to Test Case #4, however the mutation function is no longer being called.



There is no noticeable difference between the results in Test Case #4 and Test Case #5 apart from the obvious slight variances in score numbers. This is likely due to the mutation operation being a simple swap which hardly affects the layout.

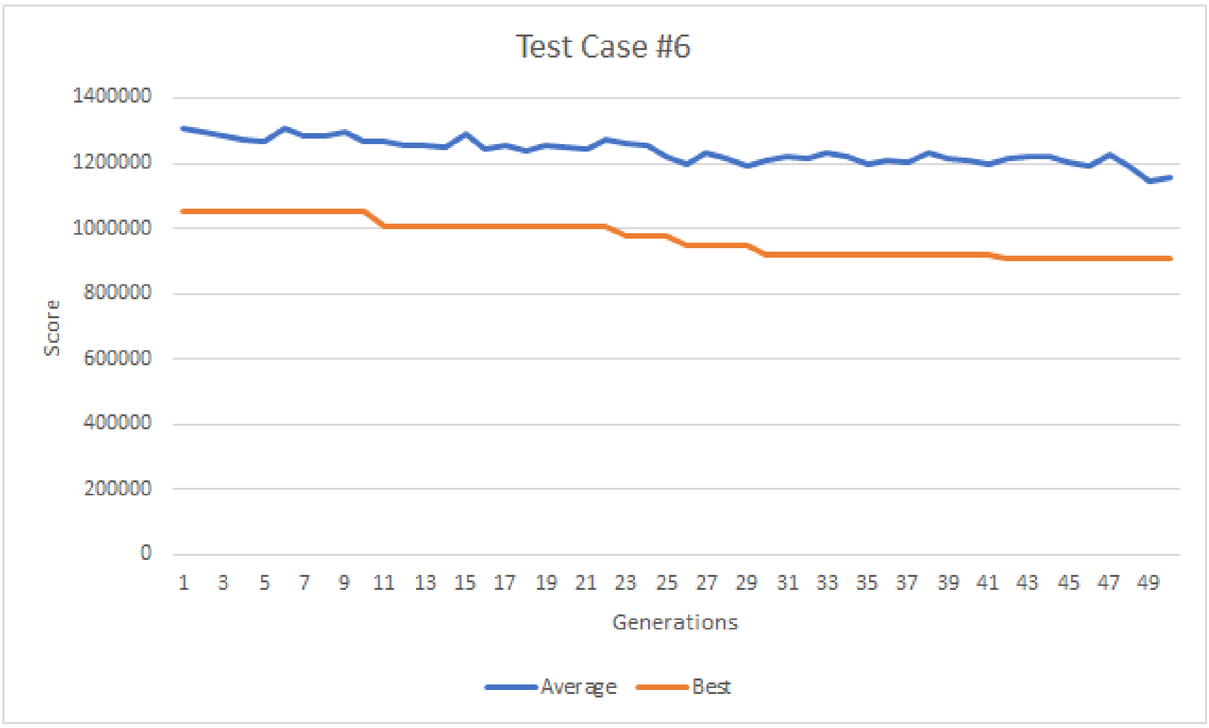
The best layout generated by this Genetic Algorithm with a score of 1,014,940 is the following:



<u>Test Case #6</u>

Population Size: 50
Number of Generations: 50
Keyboards to carry over via Elitism: 10
Mutation: No

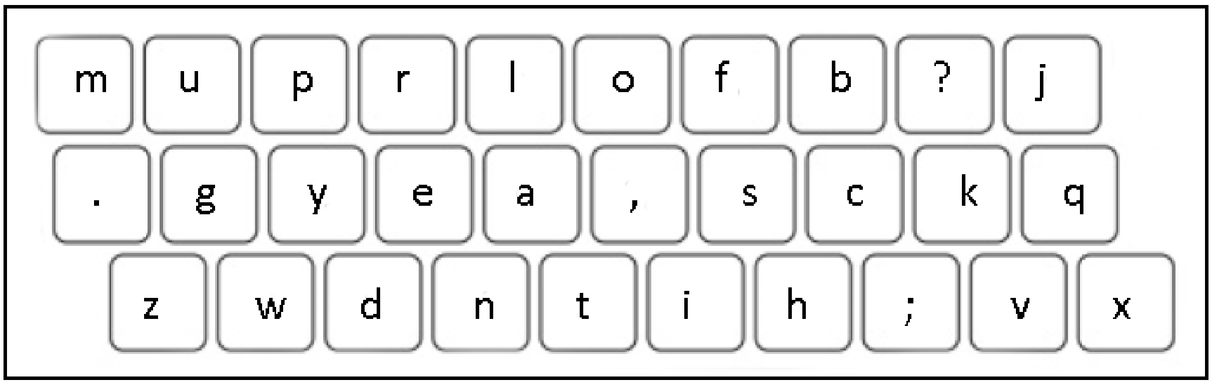
Similarly to Test Case #1 and Test Case #2, the only difference between Test Case #5 and Test Case #6 is that the number of elite layouts was doubled. This once again means that a lower average is expected more consistently.



The average scores in Test Case #4, while consistent as expected, are also much lower than anything in previous cases as a low scoring elite layout was quickly generated.

The best layout kept on decreasing in generations 23, 26, 30 and 42 as can be seen on the graph and the average score went down proportionately.

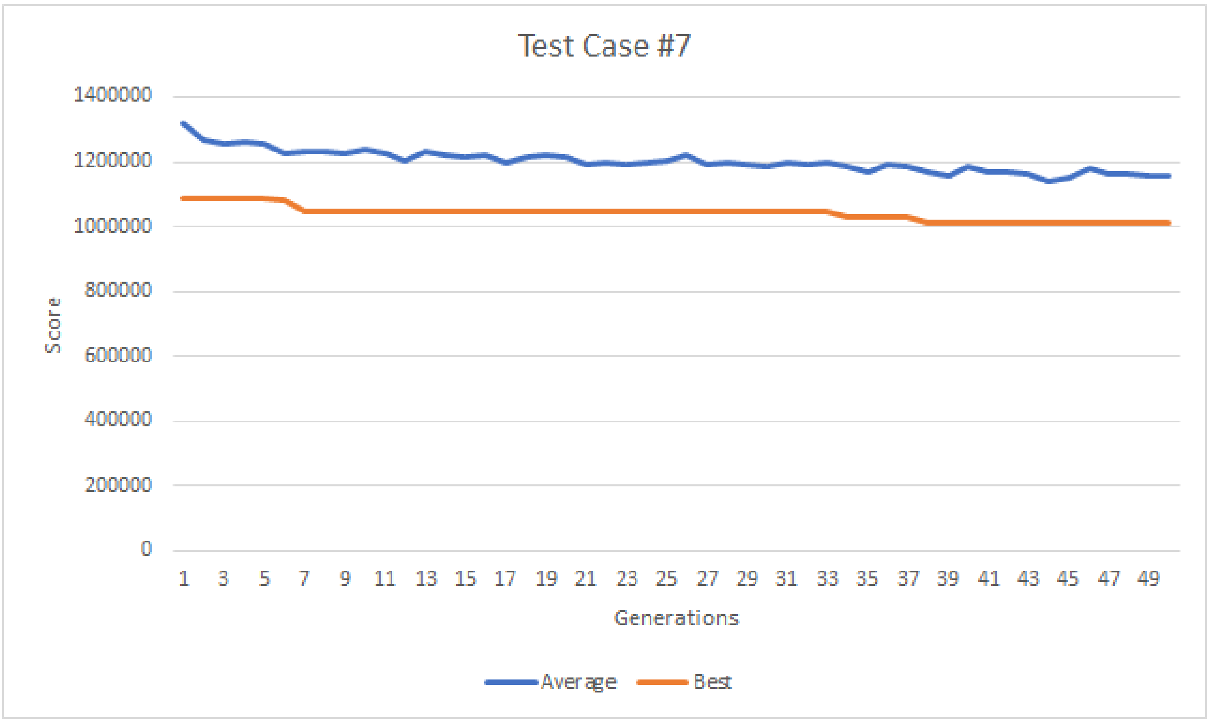
The best layout generated by this Genetic Algorithm with a score of 909,037 is the following:



Test Case #7

Population Size: 50
Number of Generations: 50
Keyboards to carry over via Elitism: 25
Mutation: Yes

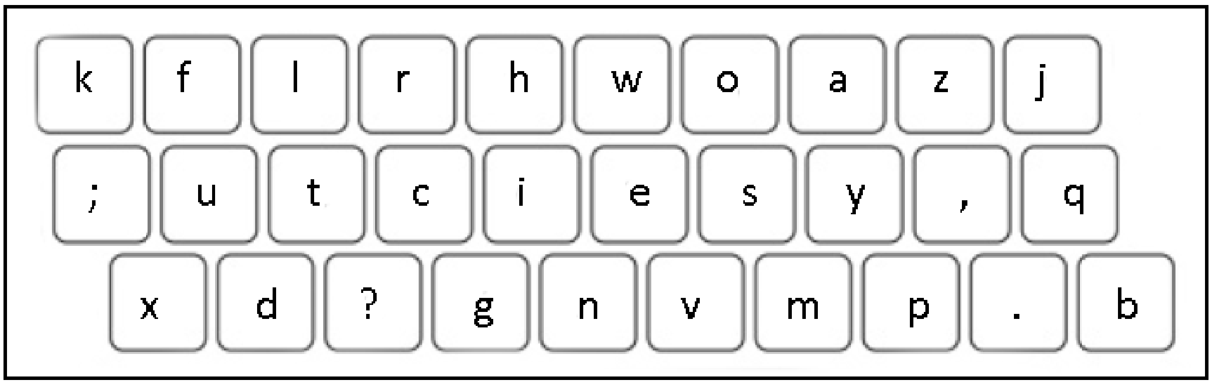
In Test Case #7, the number of elite layouts was set to half the population size. This should result in a much lower average score the more generations there are as the best half are always being saved.



As expected, Test Case #7 yielded the lowest average scores so far with scores under 1,200,000 from as early as the 17th generation.

Past the 27th generation, the 25 elite layouts were so efficient that the average score never went over 1,200,000 and even managed to consistently vary between 1,130,000 - 1,160,000 for the final 10 generations.

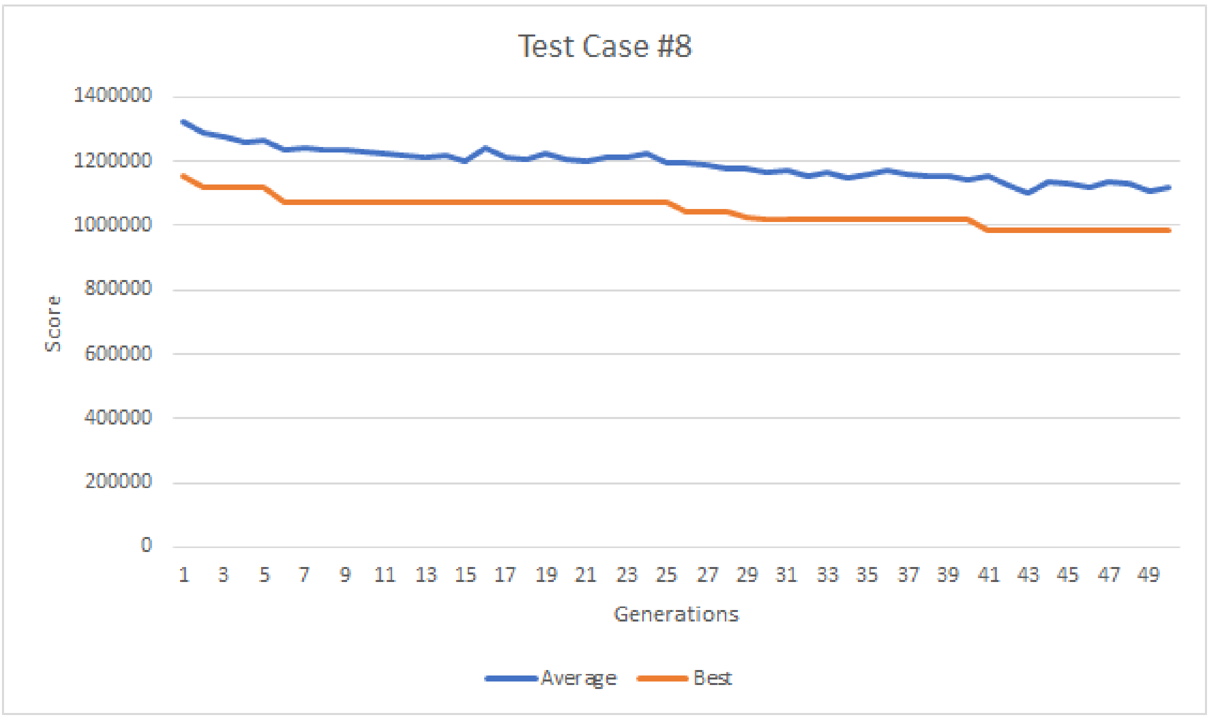
The best layout generated by this Genetic Algorithm with a score of 1,011,673 is the following:



Test Case #8

Population Size: 50
Number of Generations: 50
Keyboards to carry over via Elitism: 25
Mutation: No

Test Case #8 is another test case with half of its population being elite classes, however this time it does not use the mutation operation.

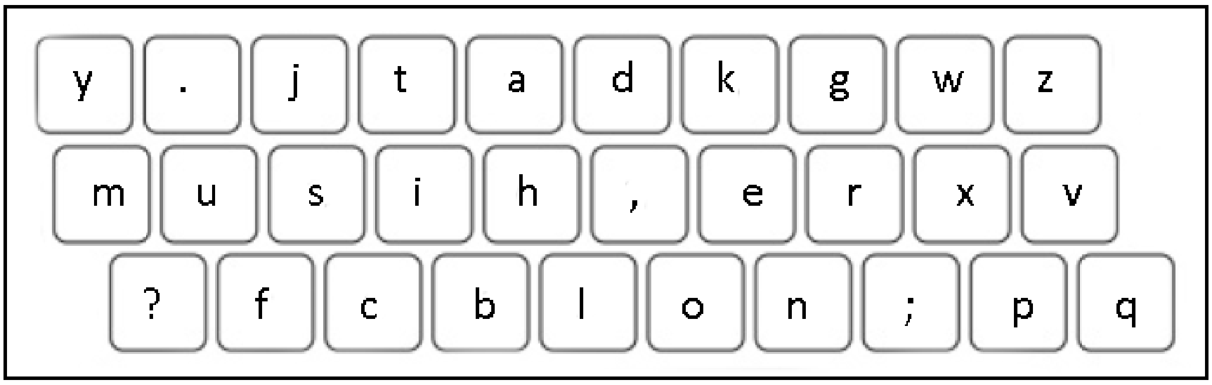


As with Test Case #4 and Test Case #5, there is not much difference when disabling the mutation operation between Test Case #7 and Test Case #8.

While the average score in Test Case #8 do get much lower in the 10 generations, the best score is also lower.

This means that better layouts were randomly generated and had Test Case #7 also had a better best score the same decrease would have occurred.

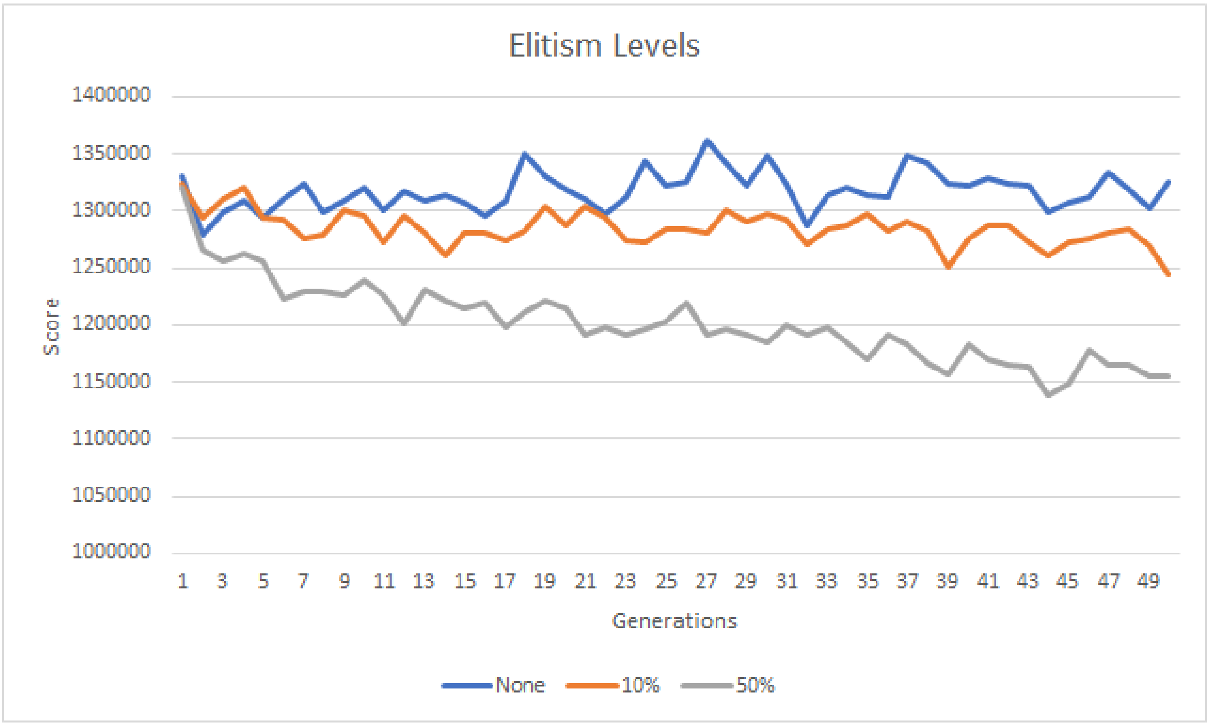
The best layout generated by this Genetic Algorithm with a score of 985,262 is the following:



Conclusion

After analysing all the Test Cases, a few interesting observations can be made.

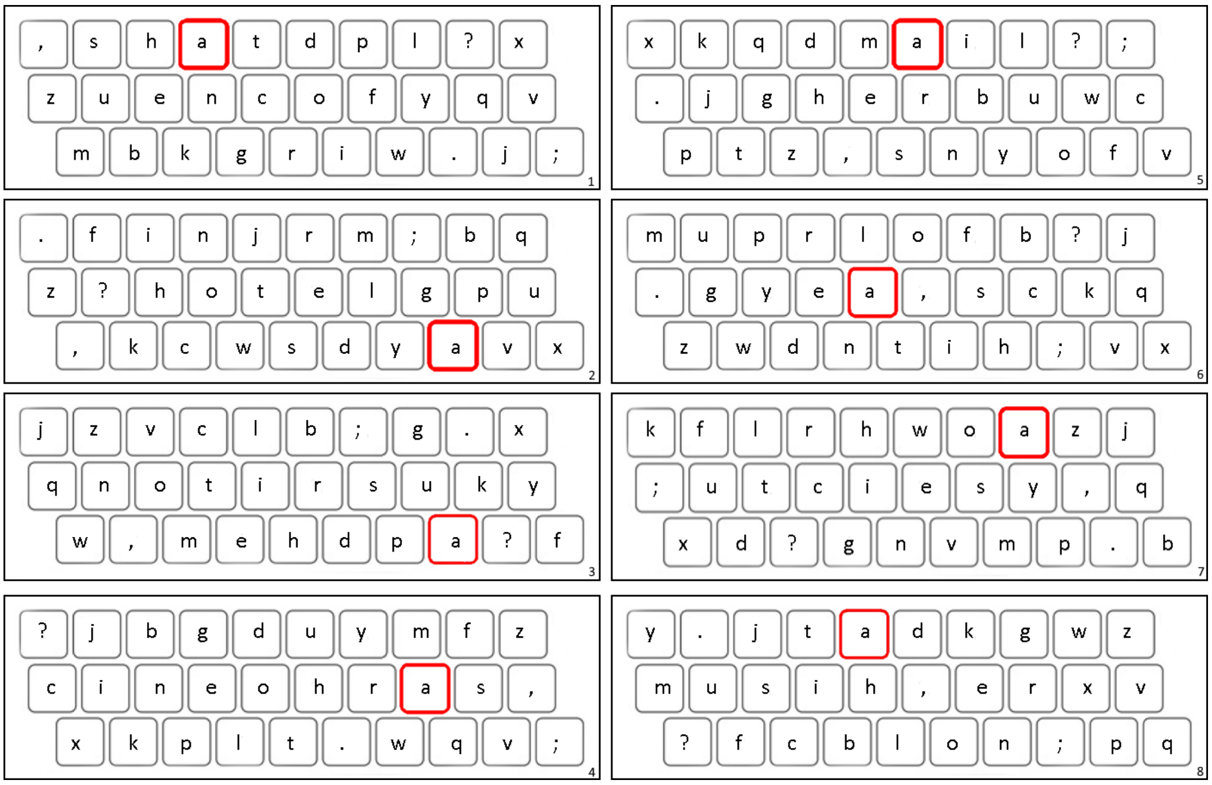
While Mutation didn't have a noticeable effect on the average scores across generations, Elitism always heavily impacted the final results. The higher the percentage of the population was saved as elite layouts, the lower the average score.



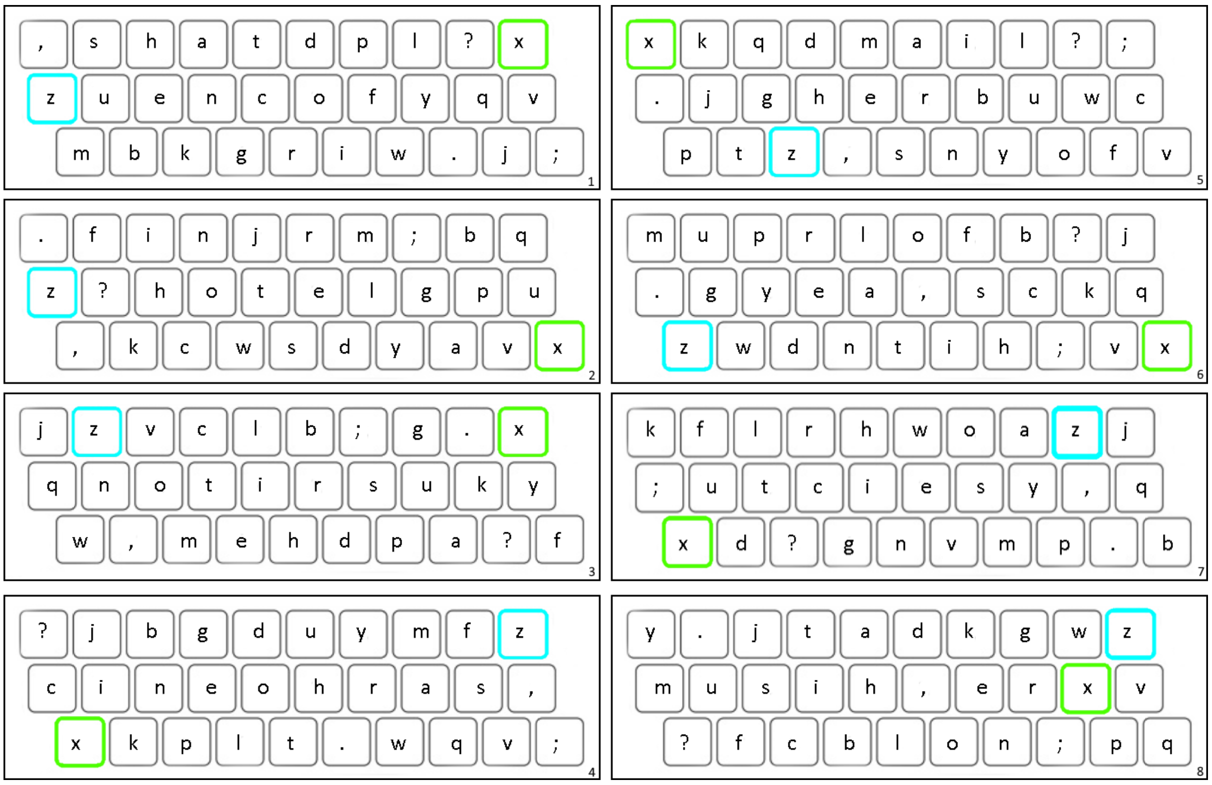
Graphs showing average scores over generations based on elitism %

As mentioned in the beginning of the evaluation, the more important characters would be in the centre of the layouts. This can be evidently seen with common characters like "a" never being more than one or two keys away from the centre.

This is in stark contrast to the characters "x" or "z", which being very rarely used elements in the corpus were always almost at corners of the layout and therefore the farthest away from the centre.

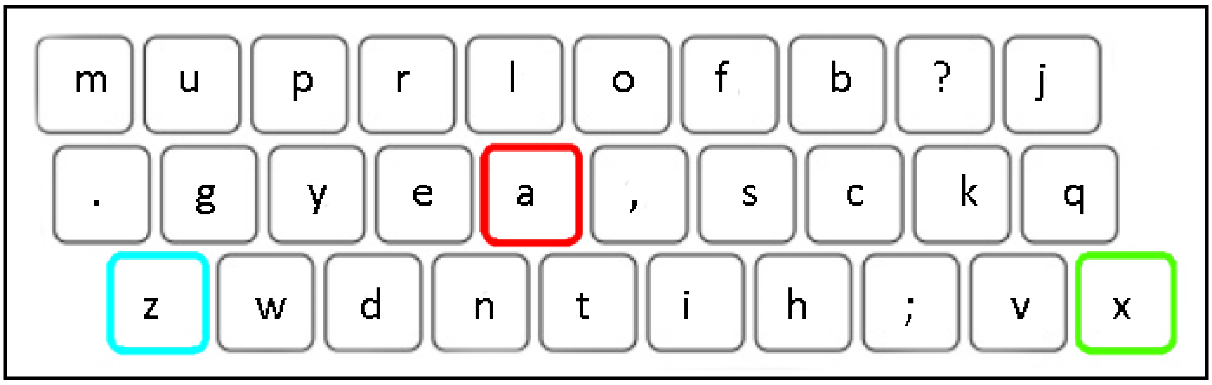


Position of "a" on all of the best layouts



Position of "x" and "z" on all the best elements

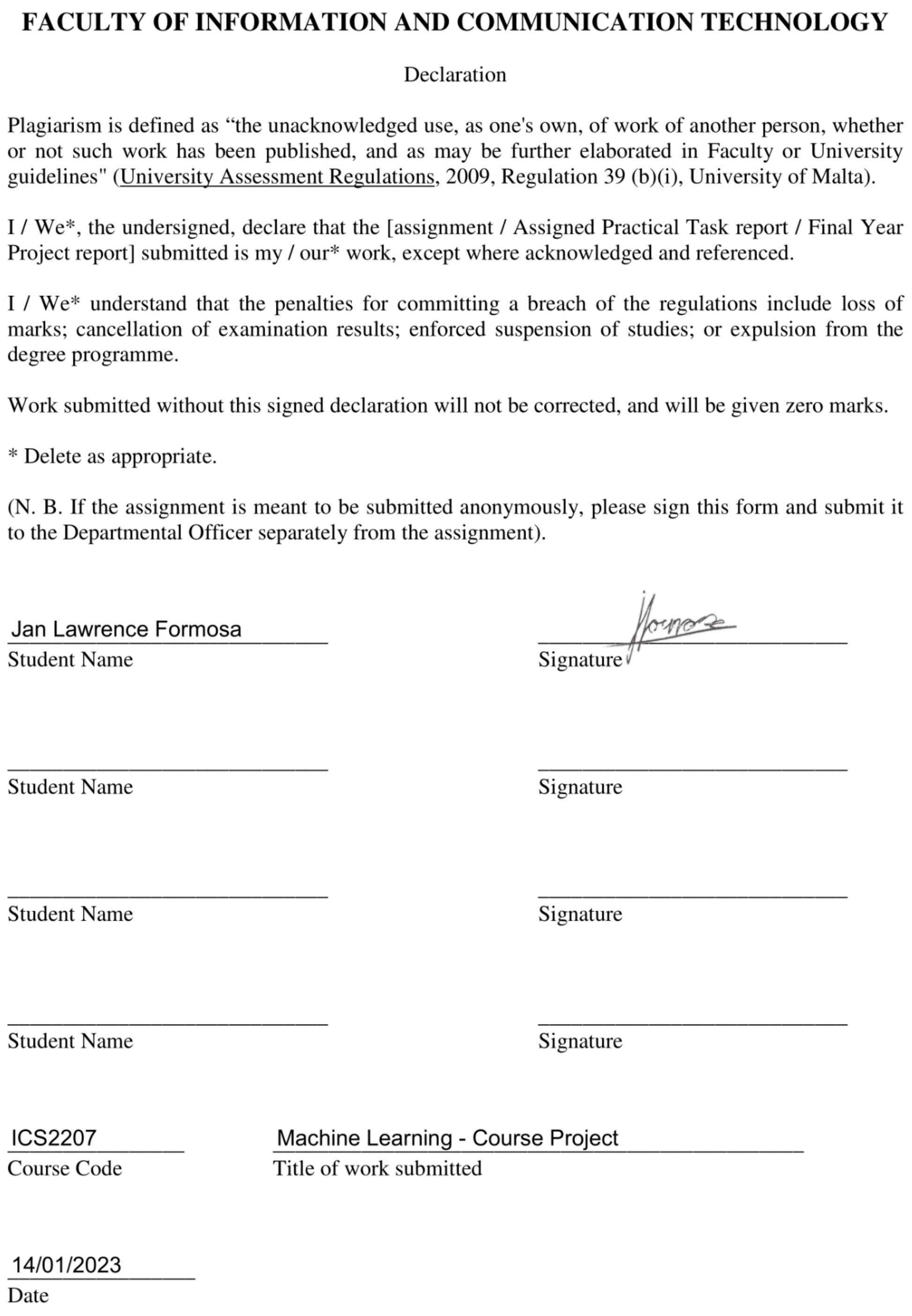
Finally the best layout scored across all the generations and test cases with a fitness score of 909,037 was in Test Case #6 and had the following layout:



Best layout from all Test Cases

In the best layout, the character “a” was one of the two centre keys whilst “x” and “z” were both on corner keys, as far away from the centre as possible. This is inline with the previous observation that the most important character will be in the centre.

Plagiarism Declaration Form



FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

Plagiarism is defined as "the unacknowledged use, as one's own, of work of another person, whether or not such work has been published, and as may be further elaborated in Faculty or University guidelines" (University Assessment Regulations, 2009, Regulation 39 (b)(i), University of Malta).

I / We*, the undersigned, declare that the [assignment / Assigned Practical Task report / Final Year Project report] submitted is my / our* work, except where acknowledged and referenced.

I / We* understand that the penalties for committing a breach of the regulations include loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected, and will be given zero marks.

* Delete as appropriate.

(N. B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).

Jan Lawrence Formosa	[illegible]
Student Name	Signature
Student Name	Signature
Student Name	Signature
Student Name	Signature

ICS2207	Machine Learning - Course Project
Course Code	Title of work submitted

14/01/2023
Date

References

- [1]“Using AI to create the perfect keyboard,” *YouTube*, 23-Sep-2022. [Online]. Available: https://youtu.be/EOaPb9wrgDY. [Accessed: 15-Jan-2023].

- [2] Forster, “A room with a view by E. M. Forster,” *Project Gutenberg*, 01-May-2001. [Online]. Available: https://www.gutenberg.org/ebooks/2641. [Accessed: 15-Jan-2023].

- [3] “Project gutenberg,” *Project Gutenberg*. [Online]. Available: https://www.gutenberg.org/. [Accessed: 15-Jan-2023].