Matthias Bartolo **0436103L**

ICS2207 - Machine Learning: Classification, Search and Optimisation (Course Project)

Introduction:

The QWERTY keyboard has become ingrained as a staple tool in the daily life of the modern man. Curious is the fact that throughout the last decade, technological innovation has skyrocketed, however such keyboard design layout has remained unchanged. Could such design be upgraded, to minimise the total distance, the fingers must move, when typing?

The main objective of this study focuses on answering the following question i.e., whether through experimentation and utilisation of AI techniques, the Perfect Keyboard could be created?

Genetic Algorithm:

A Genetic Algorithm is an algorithm which simulates the process of natural selection [1]. In other words, species who can adapt to changes in their environment, in turn can also reproduce, and thus, will be able to go to the next generation in a "survival of the fittest" based setting. The Genetic Algorithm (GA) is comprised of several **generations**, whereby, each generation consists of a **population** of **individuals**. Each individual represents a point in the problem search space, and a possible solution. Individuals are comprised of collection of data, whether it is bits, integers, character strings or float, and such data is known as a **gene**. The GA utilises a **Fitness Function** to measure the viability of an individual. Furthermore, if the individual, is not deemed, sufficiently fit, the individual, will not be given a chance to reproduce or go to the next generation.

The following study made use of this GA process, to facilitate the creation of the Perfect Keyboard. In other terms, each key on the keyboard would be a **gene**, each keyboard would be an **individual in the population (chromosome)**, the **population** would be comprised of several individual keyboards, and each **generation** would consist of a population of individuals. The study was heavily inspired from [2].

Choice of Corpus:

For this study, the text corpus utilised was: "Ben-Hur: A tale of the Christ by Lew Wallace", to discover, the most optimal keyboard, when typing a classical book, which included numerous biblical references. Upon some light research, it transpired, that the first publication of this book came out in 1880, only 6 years, after the invention of the QWERTY keyboard in 1874. Thus, one might wonder, what the most optimal keyboard would look like, when creating, such a book. The book was obtained online from [3].

Technical Aspects:

Chromosome Design:

For this study, the keyboards constructed would be in the form of a 1d character string array, and would only include 30 keys. This was done as to simplify the representation of the keyboard, and would thus, make computation much faster. The keyboards would consist of the following characters: "q", "w", "e", "r", "t", "y", "u", "i", "o", "p", "a", "s", "d", "f", "g", "h", "j", "k", "l","\"", "z", "x", "c", "v", "b", "n", "m","." ,",", ";" . The spacebar key was not included in this representation, as in real life situations, the thumbs, would always be laying on such keyboard, and thus the distance for the spacebar, would always be 0. This can be seen in Figure 1.

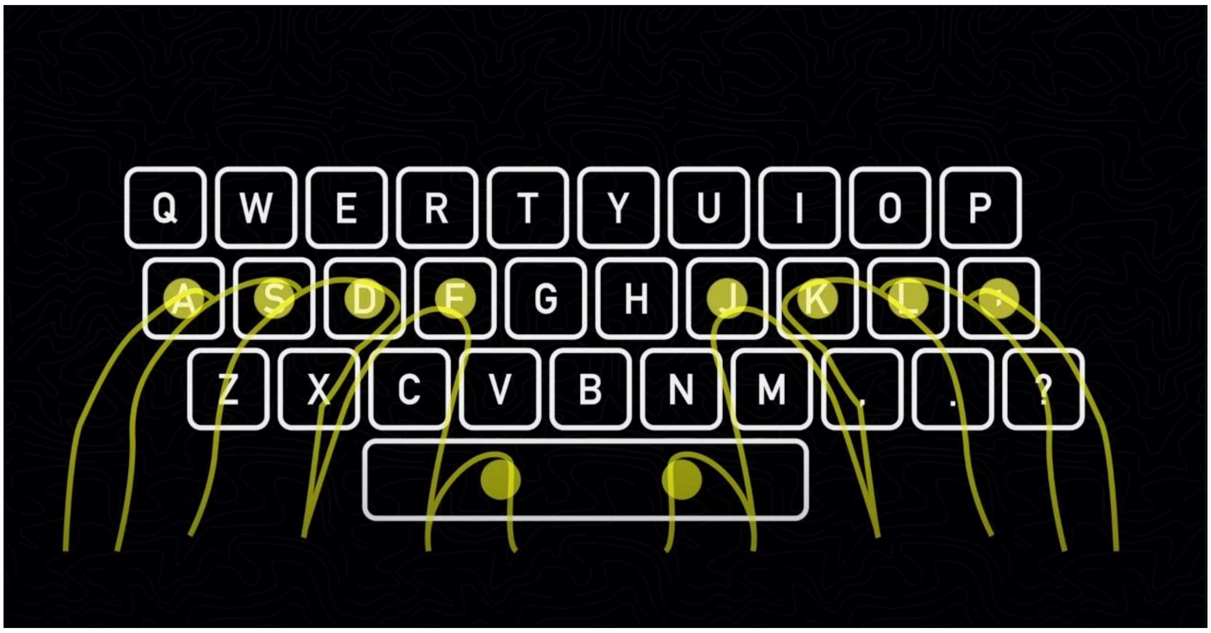


Figure 1: Visual Representation of fingers on keyboard, taken from [2]

Loading Dataset (Text Corpus):

A function was created to load and filter the text corpus utilised, and save it in a single variable, which will be used as a dataset. The filtering aspect of this function pertains, since the constructed keyboard would only, include 30 characters, and thus, the document could be comprised of other unwanted characters.

Visually Showing Keyboard (Show Keyboard):

A function was created to visually map the keys of the chromosome design onto a blank keyboard image, in order to represent the true essence of the keyboard. The function utilised the blank keyboard image retrieved from [4], as well as the font retrieved from [5]. Through some trial and error, mapping such keys on the keyboard image dynamically, was achieved by varying the x and y offset for each key, depending on the row and column of the specified key. The result of such mapping can be seen in figure 2.

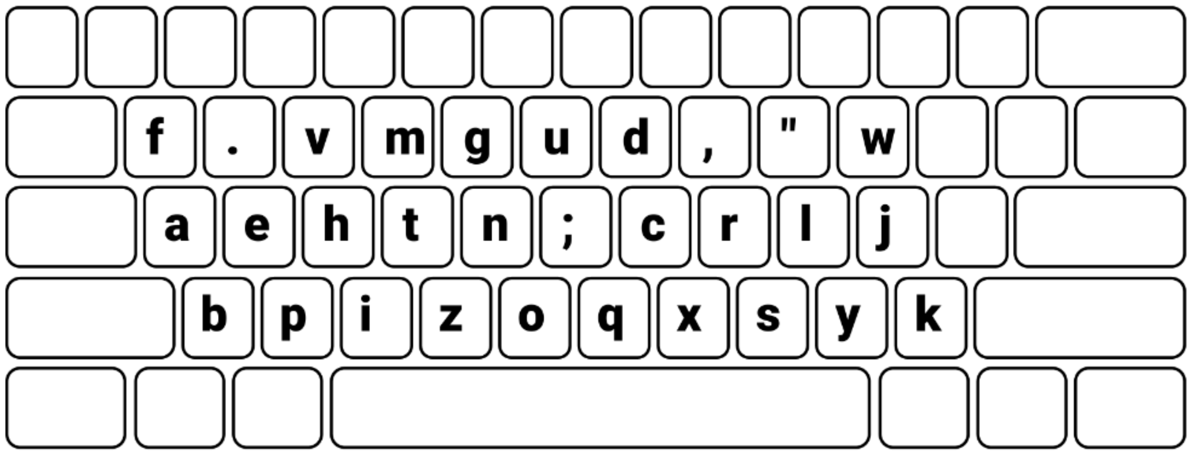


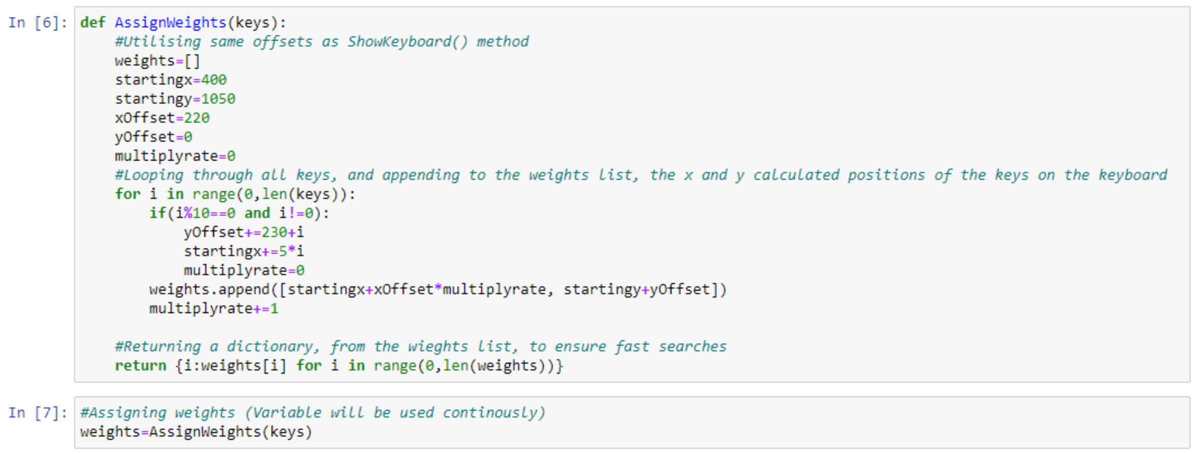
Figure 2: Visual Representation of chromosome keys on keyboard image

Fitness Function:

A function was created in order, to calculate the total distance of a given keyboard, with respect to the text corpus used. Such function required the creation of 2 smaller functions, which are the following:

1. Assign Weights Function:

Such function would utilise the same logic as the Show Keyboard function, however, instead of displaying the keyboard, it would save the x and y position of that keyboard in a dictionary, which be used later, to calculate the distance. The keys for the dictionary would be the actual key locations of the chromosome design, for instance in Figure 1, the key "f" is stored at index 0 in the chromosome design, and thus it would have x position of 400 and y position of 1050. The key at index 1 would have different offsets and so on. Design of such function can be seen in Figure 3.

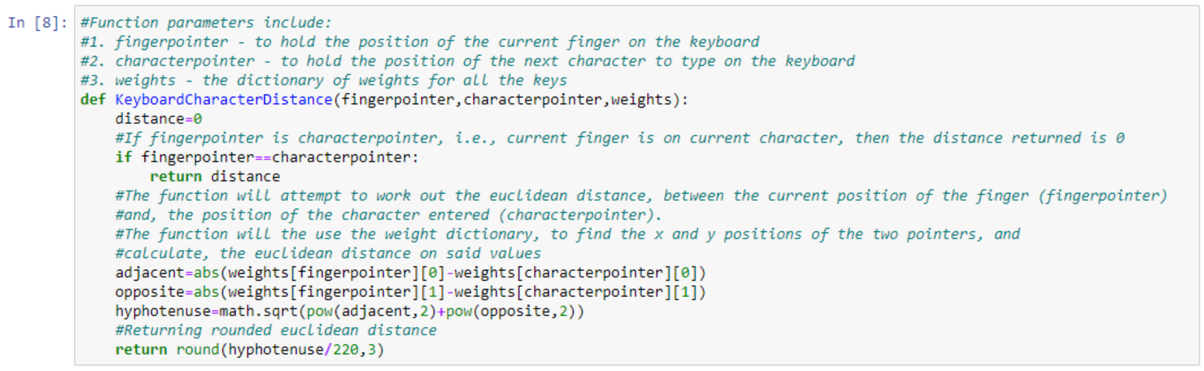


```
In [6]: def AssignWeights(keys):
            #Utilising same offsets as ShowKeyboard() method
            weights=[]
            startingx=400
            startingy=1050
            xOffset=220
            yOffset=0
            multiplyrate=0
            #Looping through all keys, and appending to the weights list, the x and y calculated positions of the keys on the keyboard
            for i in range(0,len(keys)):
                if(i%10==0 and i!=0):
                    yOffset+=230+i
                    startingx+=5*i
                    multiplyrate=0
                weights.append([startingx+xOffset*multiplyrate, startingy+yOffset])
                multiplyrate+=1

            #Returning a dictionary, from the wieghts list, to ensure fast searches
            return {i:weights[i] for i in range(0,len(weights))}

In [7]: #Assigning weights (Variable will be used continously)
        weights=AssignWeights(keys)
```

Figure 3: Design of Assign Weights function

2. Keyboard Character Distance Function:

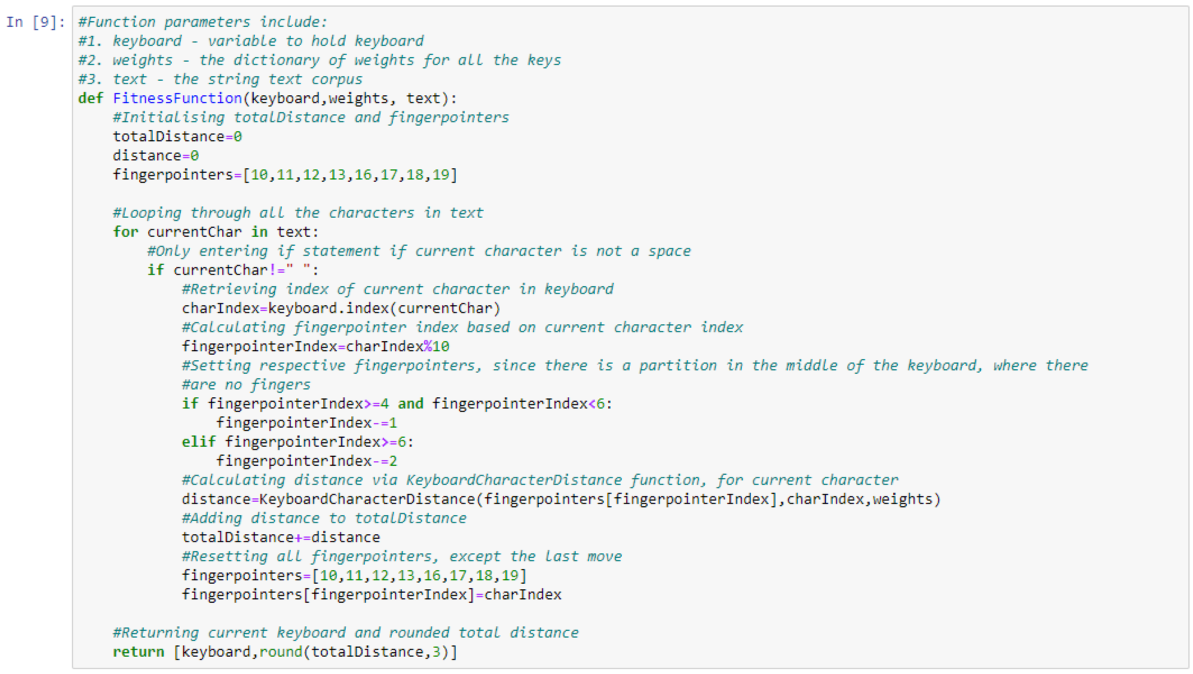
Such function would utilise the dictionary of weights populated in the Assign Weights Function and would calculate the Euclidean distance to move the current finger position on the keyboard, to next character which is typed. Euclidean distance is calculated based on the x and y positions of current key which the finger rests on, and key which the finger will move to when typing the new character. Apart from the dictionary of weights, function also takes parameters of the finger pointer, and character pointer, which will be used as indexes in the weights dictionary to retrieve the x and y positions of each pointer. The function first checks, whether the two pointers are the same, if so, then a distance of zero is recorded, if not the Euclidean distance between x and y positions, will be calculated, and the normalised distance is returned. Design of such function can be seen in Figure 4.



```
In [8]: #Function parameters include:
        #1. fingerpointer - to hold the position of the current finger on the keyboard
        #2. characterpointer - to hold the position of the next character to type on the keyboard
        #3. weights - the dictionary of weights for all the keys
        def KeyboardCharacterDistance(fingerpointer,characterpointer,weights):
            distance=0
            #If fingerpointer is characterpointer, i.e., current finger is on current character, then the distance returned is 0
            if fingerpointer==characterpointer:
                return distance
            #The function will attempt to work out the euclidean distance, between the current position of the finger (fingerpointer)
            #and, the position of the character entered (characterpointer).
            #The function will the use the weight dictionary, to find the x and y positions of the two pointers, and
            #calculate, the euclidean distance on said values
            adjacent=abs(weights[fingerpointer][0]-weights[characterpointer][0])
            opposite=abs(weights[fingerpointer][1]-weights[characterpointer][1])
            hyphotenuse=math.sqrt(pow(adjacent,2)+pow(opposite,2))
            #Returning rounded euclidean distance
            return round(hyphotenuse/220,3)
```

Figure 4: Design of Keyboard Character Distance function

With the creation of such methods, the Fitness Function could finally be built. Such function would work by saving all eight finger (excluding thumbs) positions on the keyboard chromosome design, into an array of finger pointers. The function will then loop, through all the characters in the text corpus, one by one, ignoring the space character, as it was assumed in the beginning, that such a character would have a distance of zero. If character is not a space, then function will attempt, through modulus to retrieve the column position of that character in the keyboard. In this case, the function is performing modulus by ten, since keyboard has a size of three rows by ten columns. Next, based on the column position, a small calculation had to be made to determine which finger to move. In continuation, the Keyboard Character Distance function was called, given the current finger position which needs to move, the character to enter and the weights dictionary, together with the distance of return, by such function. The distance of return is added to the total distance, and finger pointers are all reset, except the last move. The latter was done, as to mimic how a person would normally behave, when typing a character. Such function would return a list, whereby the first element would be the keyboard entered, and the second element, would be the total distance of that keyboard, rounded to three decimal points. Design of such function can be seen in Figure 5.



```
In [9]: #Function parameters include:
        #1. keyboard - variable to hold keyboard
        #2. weights - the dictionary of weights for all the keys
        #3. text - the string text corpus
        def FitnessFunction(keyboard,weights, text):
            #Initialising totalDistance and fingerpointers
            totalDistance=0
            distance=0
            fingerpointers=[10,11,12,13,16,17,18,19]

            #Looping through all the characters in text
            for currentChar in text:
                #Only entering if statement if current character is not a space
                if currentChar!=" ":
                    #Retrieving index of current character in keyboard
                    charIndex=keyboard.index(currentChar)
                    #Calculating fingerpointer index based on current character index
                    fingerpointerIndex=charIndex%10
                    #Setting respective fingerpointers, since there is a partition in the middle of the keyboard, where there
                    #are no fingers
                    if fingerpointerIndex>=4 and fingerpointerIndex<6:
                        fingerpointerIndex-=1
                    elif fingerpointerIndex>=6:
                        fingerpointerIndex-=2
                    #Calculating distance via KeyboardCharacterDistance function, for current character
                    distance=KeyboardCharacterDistance(fingerpointers[fingerpointerIndex],charIndex,weights)
                    #Adding distance to totalDistance
                    totalDistance+=distance
                    #Resetting all fingerpointers, except the last move
                    fingerpointers=[10,11,12,13,16,17,18,19]
                    fingerpointers[fingerpointerIndex]=charIndex

            #Returning current keyboard and rounded total distance
            return [keyboard,round(totalDistance,3)]
```

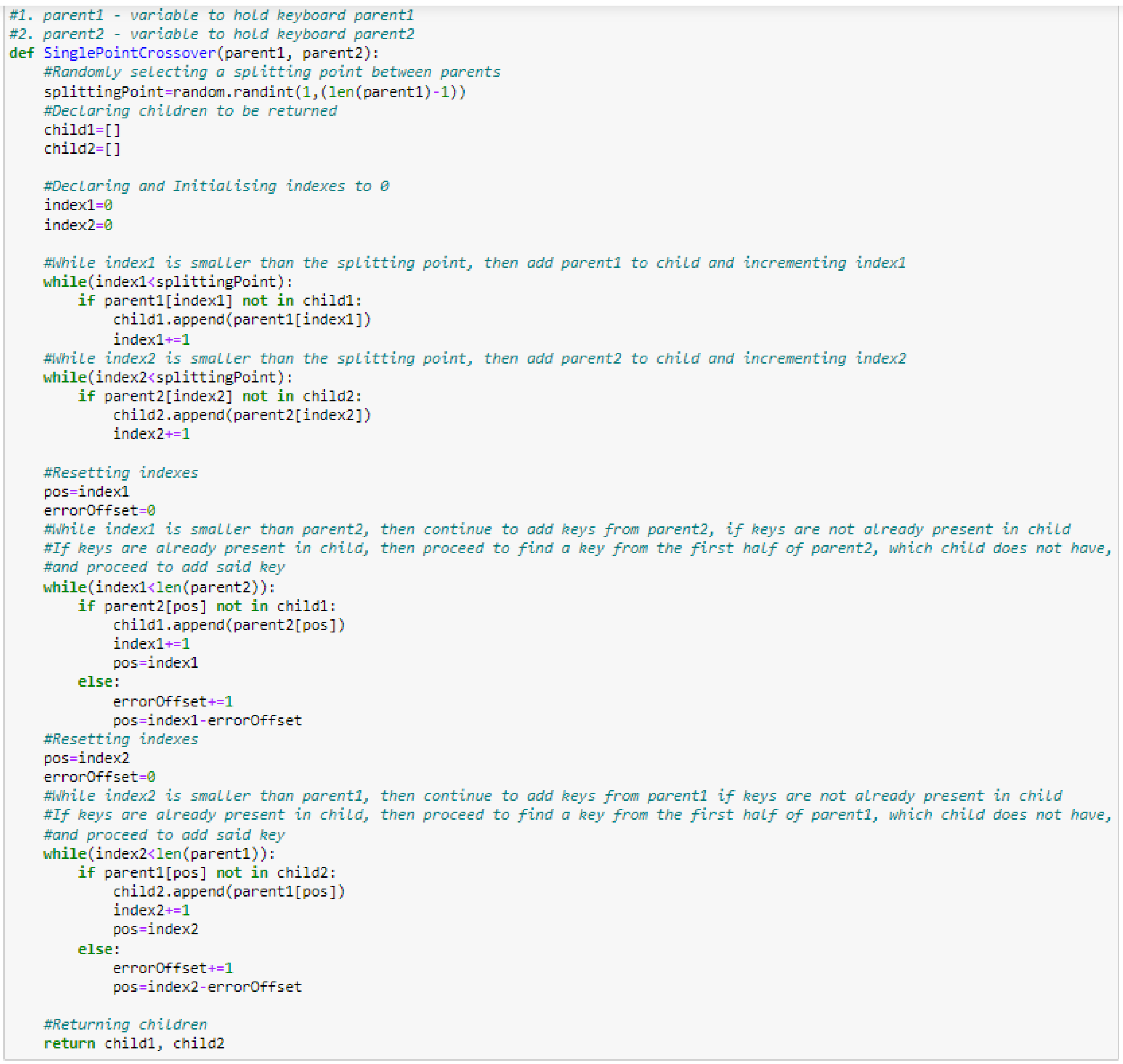
Figure 5: Design of Fitness Function function

<u>**Single Point Crossover:**</u>

A function was created in order, to perform Single Point Crossover, between two parents i.e., combining both parents, to produce two children. Rather, than following, the approach described in [2], whereby, the author describes an operation of taking only, the first vertical half of the first keyboard parent, and the remaining keys from the second keyboard parent, the study implements an operation of taking only the first horizontal half of the first keyboard and continuing to populate the child with the keys from the second keyboard parent. Instead of beginning from the starting index of the second keyboard parent and adding all the non-duplicate keys to the child, the implementation attempts to start from the splitting point index of the second keyboard parent and combining all the non-duplicate keys from the second keyboard parent to the child. If a duplicate key is found, the function will then attempt to search for a non-duplicate key in the first part of the second keyboard parent, i.e., an index which is smaller than the splitting point index. If one is discovered, the function will continue to add the keys from the second part of the second keyboard parent.

The study utilised this approach, as computation for this approach would be relatively faster than the one mentioned in [2], as well as managing to preserve the importance of the keys of the secondary keyboard. With regards, to the latter point, the function will always add to the child keyboard, the non-duplicate keys of the second keyboard parent, giving priority to the keys in the middle row of the second keyboard parent whilst proceeding in adding all the other remaining keys in either the first or third row of the second keyboard parent.

Furthermore, the splitting point between both parents, is decided randomly, from a range index: 1 and length of parent 1**.** The Function returns two children, whereby child1 would have an order of keys, starting from the first keyboard parent, and continuing with keys from the second keyboard parent. Furthermore, child2 would have an order of keys, starting from the second keyboard parent, and continuing with keys from the first keyboard parent. Although the function is lengthy, it has a time complexity of O(n), where n is the size of one of the parent's keyboards. Design of such function can be seen in Figure 6.

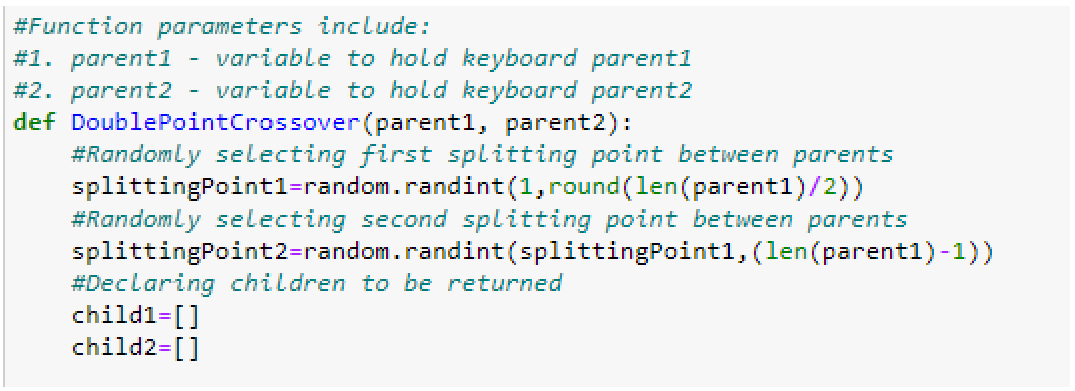


```python
#1. parent1 - variable to hold keyboard parent1
#2. parent2 - variable to hold keyboard parent2
def SinglePointCrossover(parent1, parent2):
    #Randomly selecting a splitting point between parents
    splittingPoint=random.randint(1,(len(parent1)-1))
    #Declaring children to be returned
    child1=[]
    child2=[]

    #Declaring and Initialising indexes to 0
    index1=0
    index2=0

    #While index1 is smaller than the splitting point, then add parent1 to child and incrementing index1
    while(index1<splittingPoint):
        if parent1[index1] not in child1:
            child1.append(parent1[index1])
            index1+=1
    #While index2 is smaller than the splitting point, then add parent2 to child and incrementing index2
    while(index2<splittingPoint):
        if parent2[index2] not in child2:
            child2.append(parent2[index2])
            index2+=1

    #Resetting indexes
    pos=index1
    errorOffset=0
    #While index1 is smaller than parent2, then continue to add keys from parent2, if keys are not already present in child
    #If keys are already present in child, then proceed to find a key from the first half of parent2, which child does not have,
    #and proceed to add said key
    while(index1<len(parent2)):
        if parent2[pos] not in child1:
            child1.append(parent2[pos])
            index1+=1
            pos=index1
        else:
            errorOffset+=1
            pos=index1-errorOffset
    #Resetting indexes
    pos=index2
    errorOffset=0
    #While index2 is smaller than parent1, then continue to add keys from parent1 if keys are not already present in child
    #If keys are already present in child, then proceed to find a key from the first half of parent1, which child does not have,
    #and proceed to add said key
    while(index2<len(parent1)):
        if parent1[pos] not in child2:
            child2.append(parent1[pos])
            index2+=1
            pos=index2
        else:
            errorOffset+=1
            pos=index2-errorOffset

    #Returning children
    return child1, child2
```

Figure 6: Design of Single Point Crossover function

<u>Double Point Crossover:</u>

A function was created in order, to perform Double Point Crossover, between two parents i.e., combining both parents, to produce two children. The function builds upon Single Point Crossover Function, however, instead of having one splitting point, this function has two. Furthermore, the first splitting point between both parents, is decided randomly, from a range index: 1 and length of parent/2**.** The second splitting point between both parents, is decided randomly, from a range index: first splitting point and length of parent-1. The latter points can be seen in Figure 7.



```python
#Function parameters include:
#1. parent1 - variable to hold keyboard parent1
#2. parent2 - variable to hold keyboard parent2
def DoublePointCrossover(parent1, parent2):
    #Randomly selecting first splitting point between parents
    splittingPoint1=random.randint(1,round(len(parent1)/2))
    #Randomly selecting second splitting point between parents
    splittingPoint2=random.randint(splittingPoint1,(len(parent1)-1))
    #Declaring children to be returned
    child1=[]
    child2=[]
```

Figure 7: Splitting points for Double Point Crossover

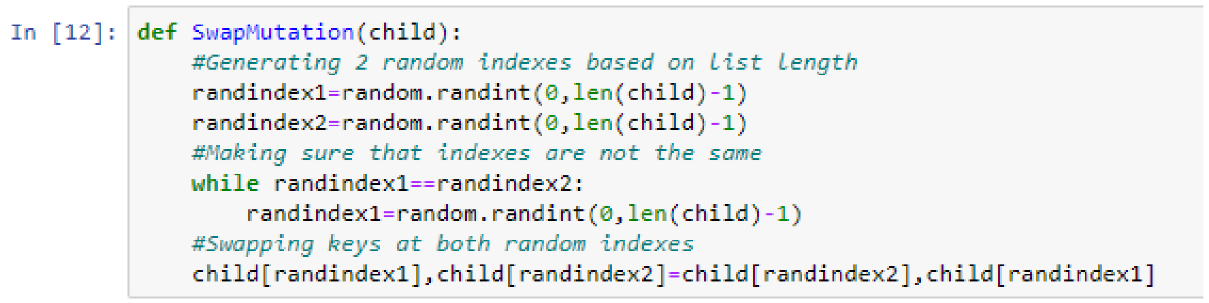
The Function returns two children, whereby child1 would have an order of keys, starting from the first keyboard parent, continuing with keys from the second keyboard parent, and ending with keys from the first keyboard parent. Furthermore, child2 would have an order of keys, starting from the second keyboard parent, continuing with keys from the first keyboard parent, and ending with keys from the second keyboard parent. Although the function is lengthy, it has a time complexity of O(n), where n is the size of one of the parents.

<u>Mutation Functions:</u>

For the purpose of this study, three mutation functions, were used, those being: Swap Mutation, Scramble Mutation and Inversion Mutation. Inspiration of such algorithms, taken from [6].

<u>1. Swap Mutation:</u>

Such function would take a child keyboard as input and would generate two random indexes of keys in the child keyboard. Function will continue to loop if indexes are the same. When two unique indexes are obtained, the Function will swap the keys at these two indexes in the child keyboard. Design of such function can be seen in Figure 8.

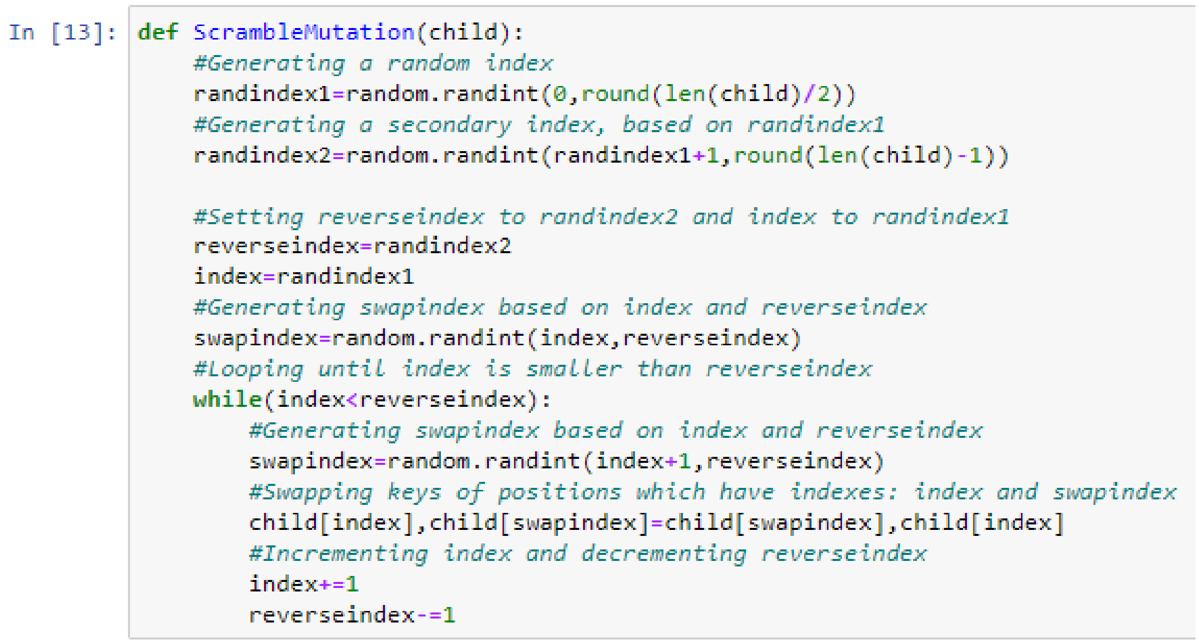


```python
In [12]: def SwapMutation(child):
             #Generating 2 random indexes based on list length
             randindex1=random.randint(0,len(child)-1)
             randindex2=random.randint(0,len(child)-1)
             #Making sure that indexes are not the same
             while randindex1==randindex2:
                 randindex1=random.randint(0,len(child)-1)
             #Swapping keys at both random indexes
             child[randindex1],child[randindex2]=child[randindex2],child[randindex1]
```

Figure 8: Design of Swap Mutation function

2. Scramble Mutation:

Such function would take a child keyboard, as input, and would generate two random indexes of keys in the child keyboard. Furthermore, index1 will be of a range from 0 to (length of child/2), and index2 will be of range from (index1+1) to (length of child-1). The function will loop for the range of index1 to index2 of the child keyboard and proceeds to generate a random swap index in the range of (index1+1) to index2. The Function will swap the keys at indexes: index1 and swap index in the child keyboard. Function will decrement index2 and increment index1. Function will also continue loop until index1 is smaller than index2. Design of such function can be seen in Figure 9.

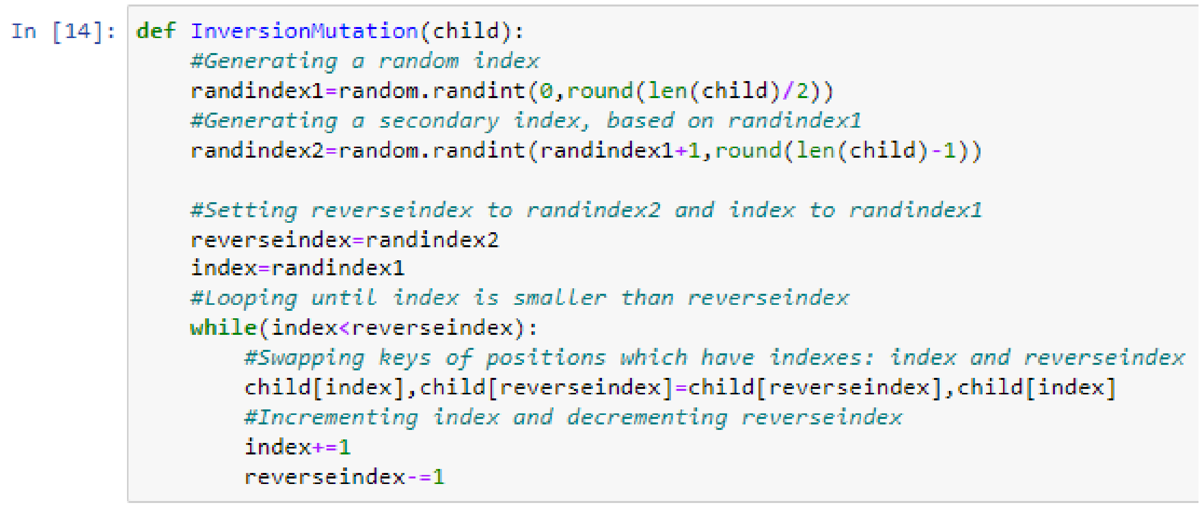


```
In [13]: def ScrambleMutation(child):
             #Generating a random index
             randindex1=random.randint(0,round(len(child)/2))
             #Generating a secondary index, based on randindex1
             randindex2=random.randint(randindex1+1,round(len(child)-1))

             #Setting reverseindex to randindex2 and index to randindex1
             reverseindex=randindex2
             index=randindex1
             #Generating swapindex based on index and reverseindex
             swapindex=random.randint(index,reverseindex)
             #Looping until index is smaller than reverseindex
             while(index<reverseindex):
                 #Generating swapindex based on index and reverseindex
                 swapindex=random.randint(index+1,reverseindex)
                 #Swapping keys of positions which have indexes: index and swapindex
                 child[index],child[swapindex]=child[swapindex],child[index]
                 #Incrementing index and decrementing reverseindex
                 index+=1
                 reverseindex-=1
```

Figure 9: Design of Scramble Mutation function

3. Inversion Mutation:

Such function would take a child keyboard as input, and would generate two random indexes of keys in the child keyboard. Furthermore, index1 will be of a range from 0 to (length of child/2), and index2 will be of range from (index1+1) to (length of child-1). The function will loop for the range of index1 to index2 of the child keyboard, and proceeds to swap the keys at indexes: index1 and index2 in the child keyboard. Function will decrement index2 and increment index1, whilst continue looping until index1 is smaller than index2. Design of such function can be seen in Figure 10.



```
In [14]: def InversionMutation(child):
             #Generating a random index
             randindex1=random.randint(0,round(len(child)/2))
             #Generating a secondary index, based on randindex1
             randindex2=random.randint(randindex1+1,round(len(child)-1))

             #Setting reverseindex to randindex2 and index to randindex1
             reverseindex=randindex2
             index=randindex1
             #Looping until index is smaller than reverseindex
             while(index<reverseindex):
                 #Swapping keys of positions which have indexes: index and reverseindex
                 child[index],child[reverseindex]=child[reverseindex],child[index]
                 #Incrementing index and decrementing reverseindex
                 index+=1
                 reverseindex-=1
```

Figure 10: Design of Inversion Mutation function

<u>The following functions, will be used by the Main Genetic Algorithm Function:</u>

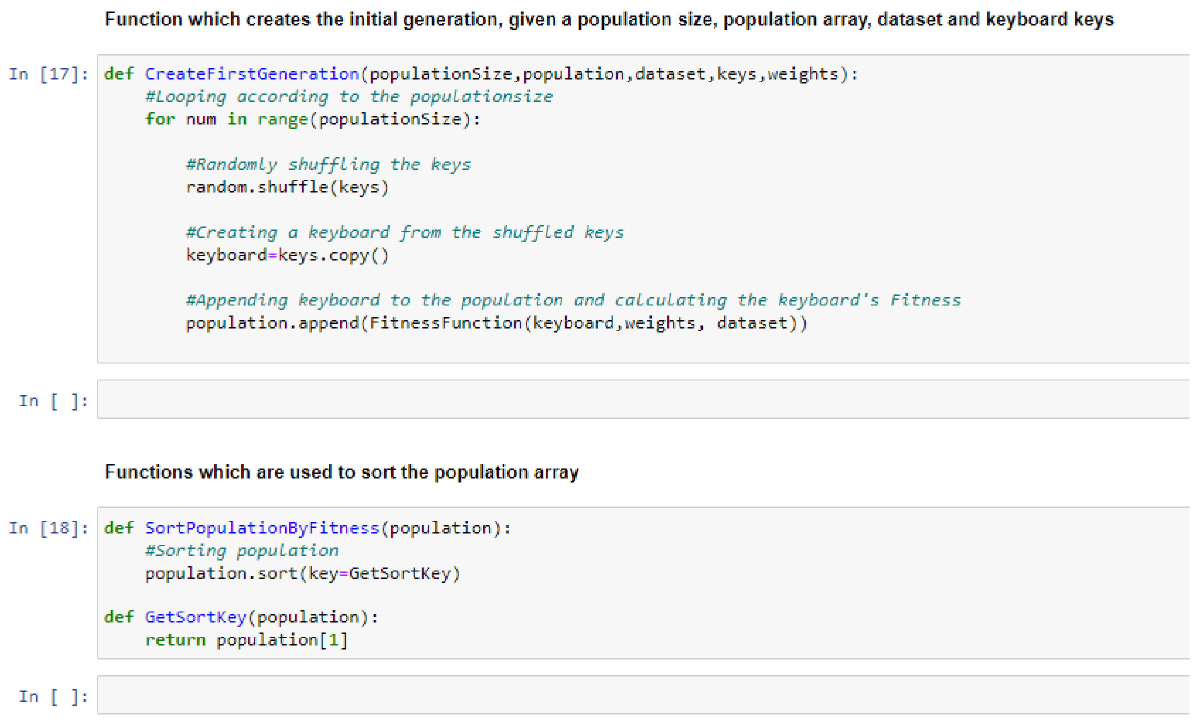
For this study, the following two functions, were utilised by the Main Genetic Algorithm Function, and can be seen in Figure 11:

<u>1. Create First Generation:</u>

Such function is responsible for the creation of the first generation of keyboard. Function works by looping according to the population size, and in every iteration, randomly shuffling the keys. Following this random combination of keys, a new keyboard is created. The new keyboard is appended to the population array, as well as it's Fitness Score.

<u>2. Sort Population by Fitness/ Get Sort Key:</u>

The Sort Population function is given the population as a parameter and proceeds to sort the population by invoking the .sort() method. Such function required the creation of another function (Get Sort Key function), which retrieves the Fitness Score of the population, and is used as a key during sorting.



Function which creates the initial generation, given a population size, population array, dataset and keyboard keys

```
In [17]: def CreateFirstGeneration(populationSize,population,dataset,keys,weights):
             #Looping according to the populationsize
             for num in range(populationSize):

                 #Randomly shuffling the keys
                 random.shuffle(keys)

                 #Creating a keyboard from the shuffled keys
                 keyboard=keys.copy()

                 #Appending keyboard to the population and calculating the keyboard's Fitness
                 population.append(FitnessFunction(keyboard,weights, dataset))

In [ ]:
```

Functions which are used to sort the population array

```
In [18]: def SortPopulationByFitness(population):
             #Sorting population
             population.sort(key=GetSortKey)

         def GetSortKey(population):
             return population[1]

In [ ]:
```

Figure 11: Design of Functions which will be used by Main Genetic Algorithm

Main Genetic Algorithm Function:

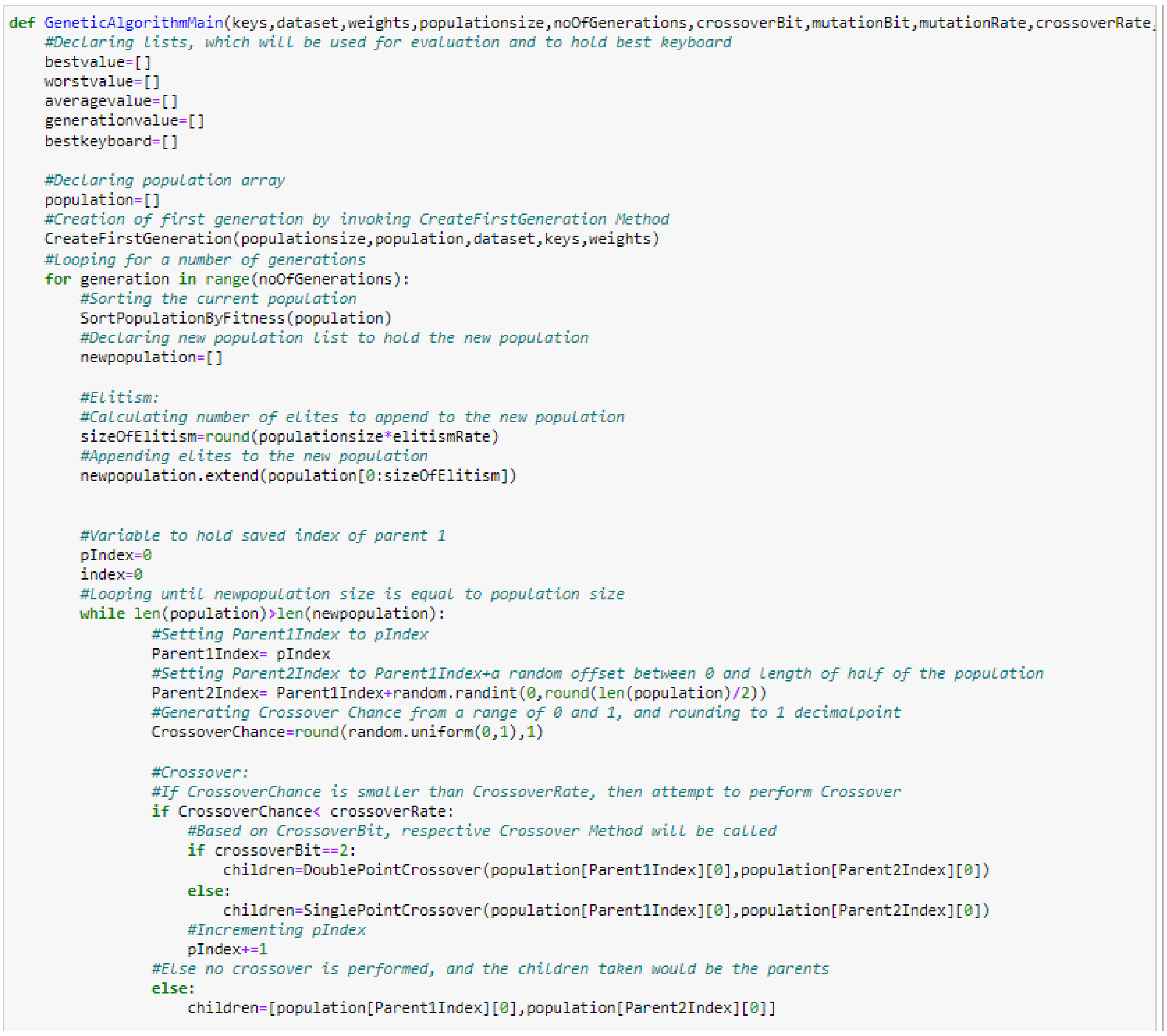
The Main Genetic Algorithm Function, combines all the previous methods, whilst taking into account the following parameters:

1. **keys** - Variable to hold the keys which will be used to create keyboards
2. **dataset** - Variable to hold the text corpus as a string
3. **weights** - Variable to hold the dictionary of weights for all the keys
4. **population Size** - Variable to hold the size of the population
5. **no Of Generations** - Variable to hold the amount of generations
6. **crossover Bit** - Variable which will be used to determine which crossover operation to perform
7. **mutation Bit** - Variable which will be used to determine which mutation operation to perform
8. **mutation Rate** - Variable which will be used to determine the rate of mutation operation
9. **crossover Rate** - Variable which will be used to determine the rate of the crossover operation
10. **elitism Rate** - Variable which will be used to determine the rate of the elitism operation.

It was decided, to construct the function in this manner, to allow numerous options of customizability, and experimentation, between the different set of values.

The Main Genetic Algorithm Function first part, as can be seen in Figure 12, works as follow:

1. First, the lists, which will be used for evaluation, as well as variable which will display the best keyboard, are declared.
2. Next, an empty population array is declared, which will hold the individuals, to be extracted from the current generation.
3. Furthermore, the first generation is created through Create First Generation function; thus, population array would be populated with random keyboards.
4. Consequently, the function loops for a number of generations, specified as a function parameter.
5. For every generation, the population is sorted by fitness, through the Sort by Fitness function.
6. Next, an empty new population array is declared, which will hold the individuals, which will move to the next generation.
7. The function then continues its operation, by calculating the number of elites, which will go to the next generation i.e., Elitism. The number of elites, which will go to the next generation, is determined, by the elitism rate, and by size of the population.
8. Furthermore, the function, proceeds, to fill the new population array with the children of the current population i.e., applying Crossover and Mutation.
9. The function continues to loop until the size of the new population is smaller, than that of the current population.
10. Consequently, the function, holds two indexes, which will be used to select, which parents will be used to perform Crossover on. The indexes are constructed in such a way, to perform Crossover on the parents with the best Fitness, as well as including a random offset, so as not to discard the possibility of performing Crossover over other unfit individuals.
11. Next, a Crossover Chance is generated from a range of zero to one, to determine the chance, whether a crossover should occur.
12. If the Crossover Chance is smaller than the Crossover Rate, then the function will proceed to perform Crossover on those two parents, otherwise no Crossover will occur, and the children are set to be the parents. Depending on the Crossover Bit, and whether there is a chance that Crossover occurs, the function will perform Double Point Crossover if bit is two-, and Single-Point Crossover otherwise.

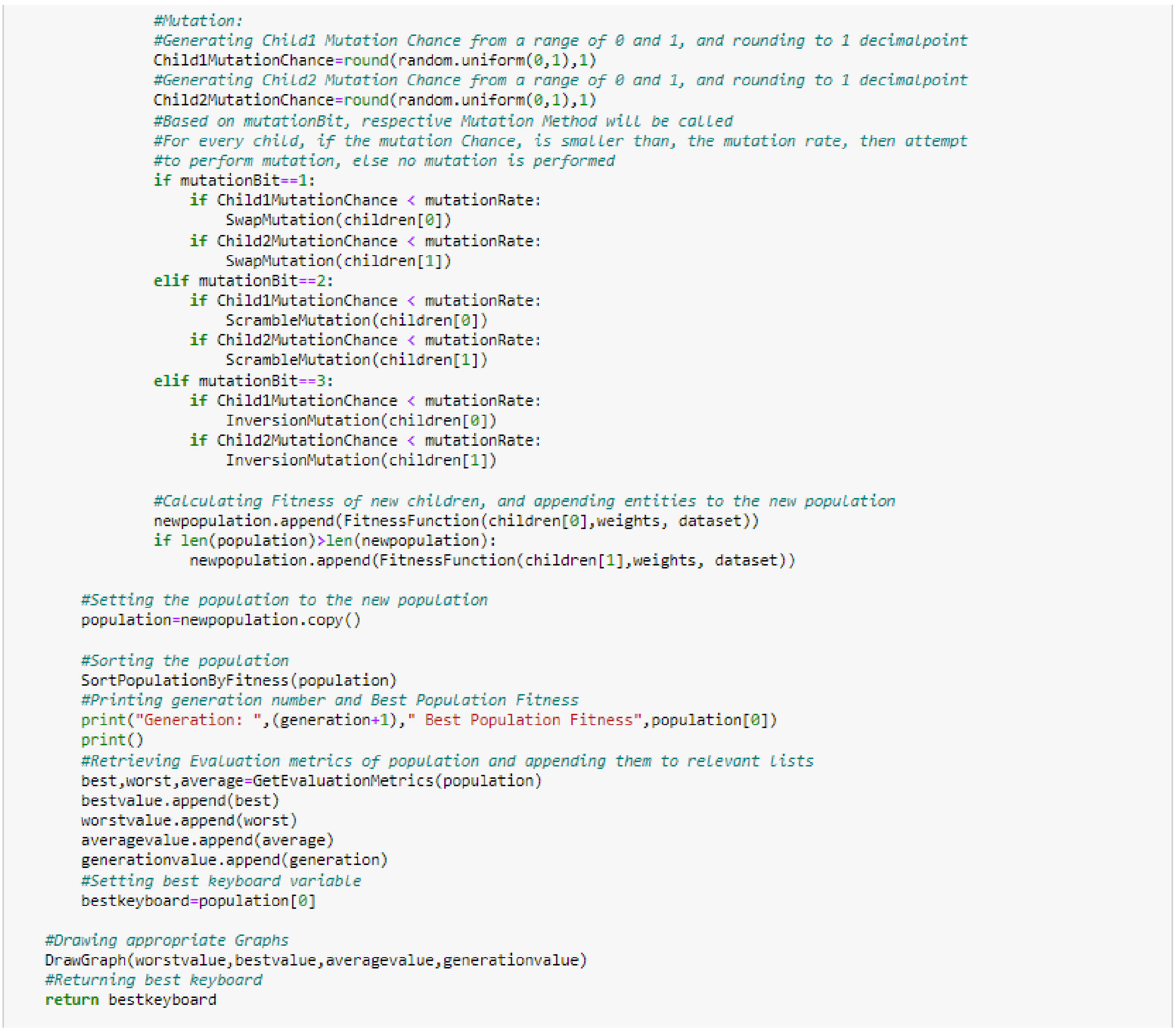


```
def GeneticAlgorithmMain(keys,dataset,weights,populationsize,noOfGenerations,crossoverBit,mutationBit,mutationRate,crossoverRate,
    #Declaring lists, which will be used for evaluation and to hold best keyboard
    bestvalue=[]
    worstvalue=[]
    averagevalue=[]
    generationvalue=[]
    bestkeyboard=[]

    #Declaring population array
    population=[]
    #Creation of first generation by invoking CreateFirstGeneration Method
    CreateFirstGeneration(populationsize,population,dataset,keys,weights)
    #Looping for a number of generations
    for generation in range(noOfGenerations):
        #Sorting the current population
        SortPopulationByFitness(population)
        #Declaring new population list to hold the new population
        newpopulation=[]

        #Elitism:
        #Calculating number of elites to append to the new population
        sizeOfElitism=round(populationsize*elitismRate)
        #Appending elites to the new population
        newpopulation.extend(population[0:sizeOfElitism])


        #Variable to hold saved index of parent 1
        pIndex=0
        index=0
        #Looping until newpopulation size is equal to population size
        while len(population)>len(newpopulation):
                #Setting Parent1Index to pIndex
                Parent1Index= pIndex
                #Setting Parent2Index to Parent1Index+a random offset between 0 and length of half of the population
                Parent2Index= Parent1Index+random.randint(0,round(len(population)/2))
                #Generating Crossover Chance from a range of 0 and 1, and rounding to 1 decimalpoint
                CrossoverChance=round(random.uniform(0,1),1)

                #Crossover:
                #If CrossoverChance is smaller than CrossoverRate, then attempt to perform Crossover
                if CrossoverChance< crossoverRate:
                    #Based on CrossoverBit, respective Crossover Method will be called
                    if crossoverBit==2:
                        children=DoublePointCrossover(population[Parent1Index][0],population[Parent2Index][0])
                    else:
                        children=SinglePointCrossover(population[Parent1Index][0],population[Parent2Index][0])
                    #Incrementing pIndex
                    pIndex+=1
                #Else no crossover is performed, and the children taken would be the parents
                else:
                    children=[population[Parent1Index][0],population[Parent2Index][0]]
```

Figure 12: Design of Main Genetic Algorithm Function part 1

The Main Genetic Algorithm Function second part, as can be seen in Figure 13, works as follow:

1. Continuing from where the first part, the function performs Mutation, on the children obtained from the Crossover operation.
2. A Mutation Chance for each child, is generated from a range of zero to one, to determine the chance, whether a mutation on that child, should occur.
3. Depending on the Mutation Bit, the function will perform Swap Mutation if bit is one-, Scramble Mutation if bit is two-, and Inversion Mutation if bit is three-. If the Mutation Chance for a child is smaller than, the Mutation Rate, then function will proceed to perform Mutation on that child, otherwise no Mutation will occur.
4. The function will then proceed to calculate the Fitness of the two new children, and append the children to the new population. A statement is being used to ensure that the new population size, would have the same size as the current population size.
5. Finally, after the new population is determined, the current population is initialised with the instances of the new population, and the former would be used to generate the next generation.
6. The current population is sorted by Fitness, through the Sort by Fitness function, and the keyboard with the Best Fitness score is displayed to the user, as well as it's Fitness Score.
7. The function calls the Get Evaluation Metrics function and stores result in the appropriate list, as well as store the best keyboard, and its Fitness.
8. After the function finishes looping for the desired number of generations, a graph is displayed, through the Draw Graph Method, to show the change in the population Fitness. The function returns the best keyboard and its Fitness



```
                #Mutation:
                #Generating Child1 Mutation Chance from a range of 0 and 1, and rounding to 1 decimalpoint
                Child1MutationChance=round(random.uniform(0,1),1)
                #Generating Child2 Mutation Chance from a range of 0 and 1, and rounding to 1 decimalpoint
                Child2MutationChance=round(random.uniform(0,1),1)
                #Based on mutationBit, respective Mutation Method will be called
                #For every child, if the mutation Chance, is smaller than, the mutation rate, then attempt
                #to perform mutation, else no mutation is performed
                if mutationBit==1:
                    if Child1MutationChance < mutationRate:
                        SwapMutation(children[0])
                    if Child2MutationChance < mutationRate:
                        SwapMutation(children[1])
                elif mutationBit==2:
                    if Child1MutationChance < mutationRate:
                        ScrambleMutation(children[0])
                    if Child2MutationChance < mutationRate:
                        ScrambleMutation(children[1])
                elif mutationBit==3:
                    if Child1MutationChance < mutationRate:
                        InversionMutation(children[0])
                    if Child2MutationChance < mutationRate:
                        InversionMutation(children[1])

                #Calculating Fitness of new children, and appending entities to the new population
                newpopulation.append(FitnessFunction(children[0],weights, dataset))
                if len(population)>len(newpopulation):
                    newpopulation.append(FitnessFunction(children[1],weights, dataset))

        #Setting the population to the new population
        population=newpopulation.copy()

        #Sorting the population
        SortPopulationByFitness(population)
        #Printing generation number and Best Population Fitness
        print("Generation: ",(generation+1)," Best Population Fitness",population[0])
        print()
        #Retrieving Evaluation metrics of population and appending them to relevant lists
        best,worst,average=GetEvaluationMetrics(population)
        bestvalue.append(best)
        worstvalue.append(worst)
        averagevalue.append(average)
        generationvalue.append(generation)
        #Setting best keyboard variable
        bestkeyboard=population[0]

    #Drawing appropriate Graphs
    DrawGraph(worstvalue,bestvalue,averagevalue,generationvalue)
    #Returning best keyboard
    return bestkeyboard
```

Figure 13: Design of Main Genetic Algorithm Function part 2

Evaluation Functions:

For this study, the following Evaluation functions, were constructed, and can be seen in Figure 14:

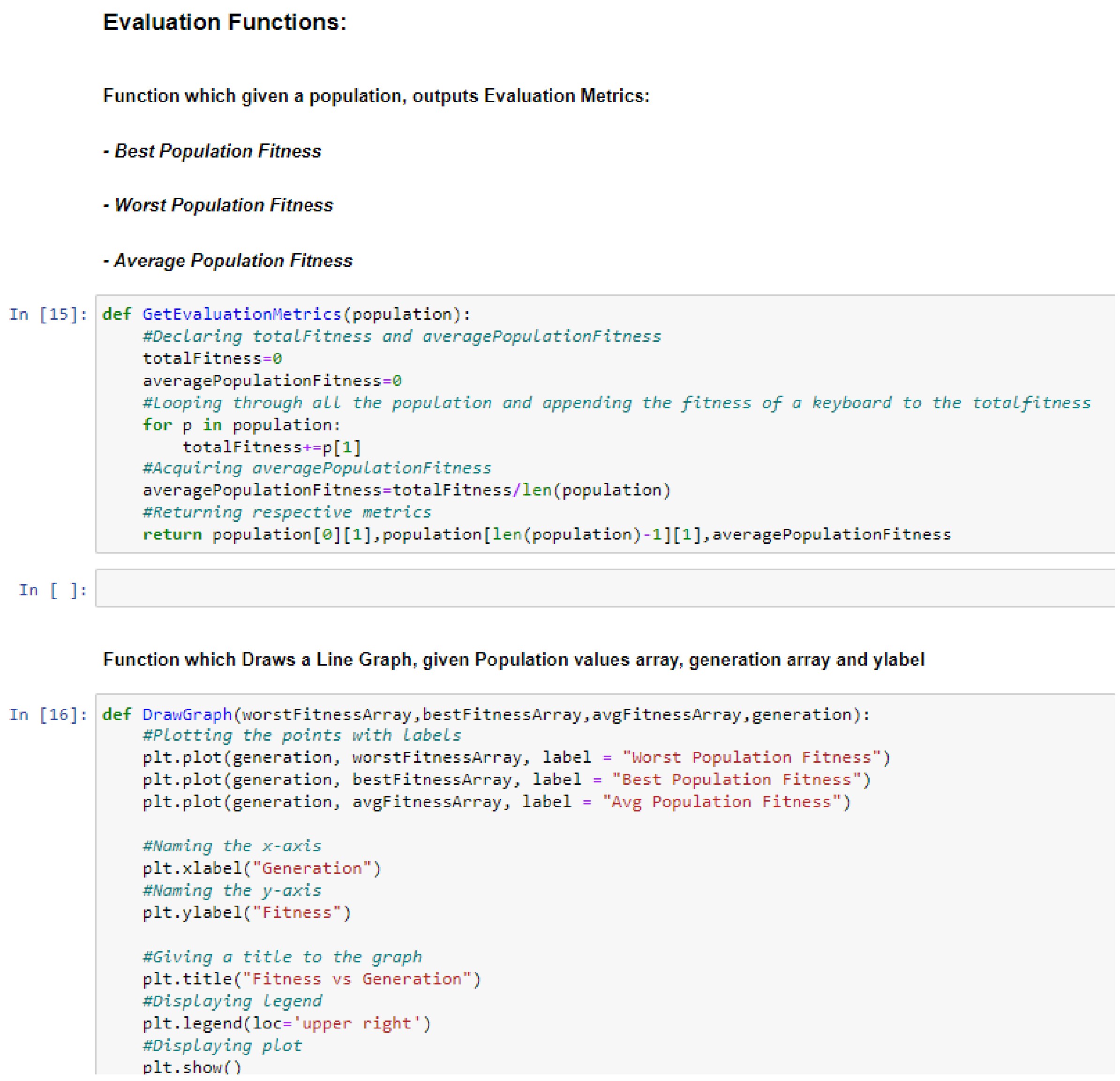
1. Get Evaluation Metrics:

Such function, given a population, would determine the Fitness values of the:

1. Keyboard, with the best Fitness
2. Keyboard, with the worst Fitness
3. The average population Fitness of the inputted population.

2. Draw Graph:

Such function, draws multiple line graphs, illustrating in a comparative manner the Best Fitness value, the Worst Fitness value, and the Average Fitness value, from the data retrieved from the Get Evaluation Metrics function, when compared to the change in generations.



Evaluation Functions:

Function which given a population, outputs Evaluation Metrics:

- Best Population Fitness

- Worst Population Fitness

- Average Population Fitness

```
In [15]: def GetEvaluationMetrics(population):
             #Declaring totalFitness and averagePopulationFitness
             totalFitness=0
             averagePopulationFitness=0
             #Looping through all the population and appending the fitness of a keyboard to the totalfitness
             for p in population:
                 totalFitness+=p[1]
             #Acquiring averagePopulationFitness
             averagePopulationFitness=totalFitness/len(population)
             #Returning respective metrics
             return population[0][1],population[len(population)-1][1],averagePopulationFitness

In [ ]:
```

Function which Draws a Line Graph, given Population values array, generation array and ylabel

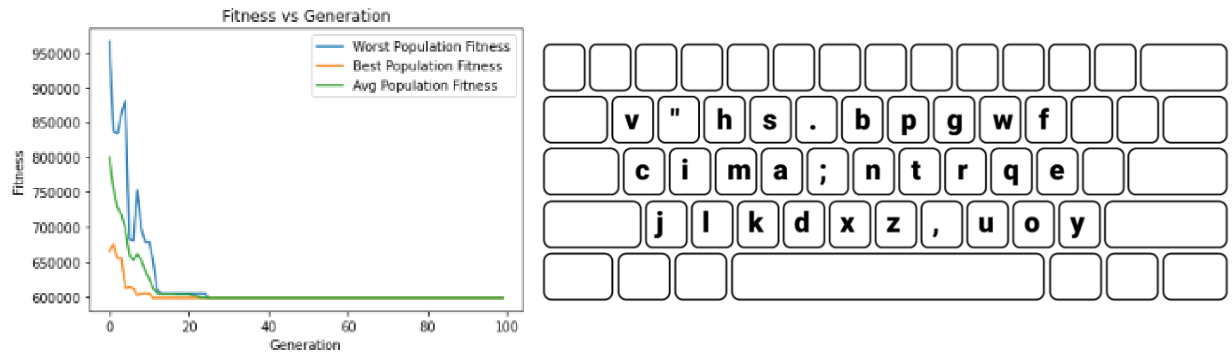
```
In [16]: def DrawGraph(worstFitnessArray,bestFitnessArray,avgFitnessArray,generation):
             #Plotting the points with labels
             plt.plot(generation, worstFitnessArray, label = "Worst Population Fitness")
             plt.plot(generation, bestFitnessArray, label = "Best Population Fitness")
             plt.plot(generation, avgFitnessArray, label = "Avg Population Fitness")

             #Naming the x-axis
             plt.xlabel("Generation")
             #Naming the y-axis
             plt.ylabel("Fitness")

             #Giving a title to the graph
             plt.title("Fitness vs Generation")
             #Displaying legend
             plt.legend(loc='upper right')
             #Displaying plot
             plt.show()
```

Figure 14: Design of Evaluation Functions

Evaluation:

The Main Genetic Algorithm function was tested for the following test cases, with the full text corpus mentioned above. Please note that the test cases 10 till 13 were inspired from the study in [7].

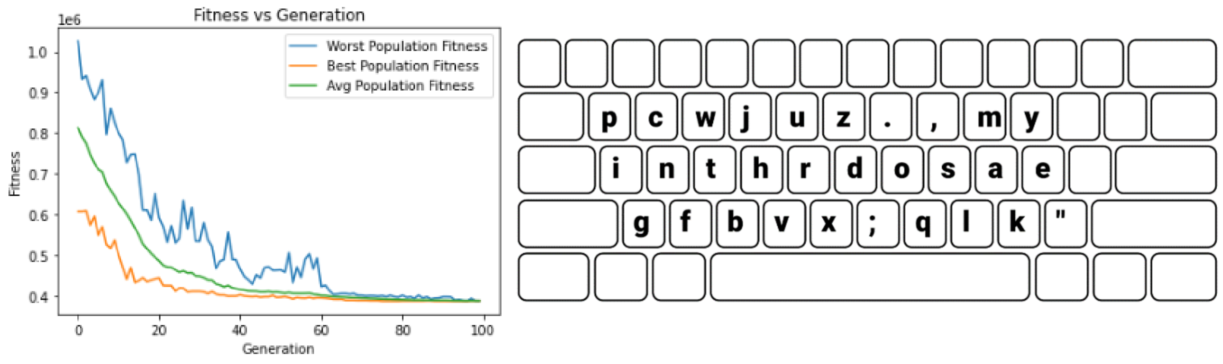
Test Case 1: Testing with a population of 10, for 100 generations, Single Point Crossover, and a crossover rate of 0.9. The results obtained (Graph showing change of population Fitness, and optimal keyboard), can be seen in Figure 15.



```
Time taken to execute:  982.97  s
```

Figure 15: Results of Test Case 1

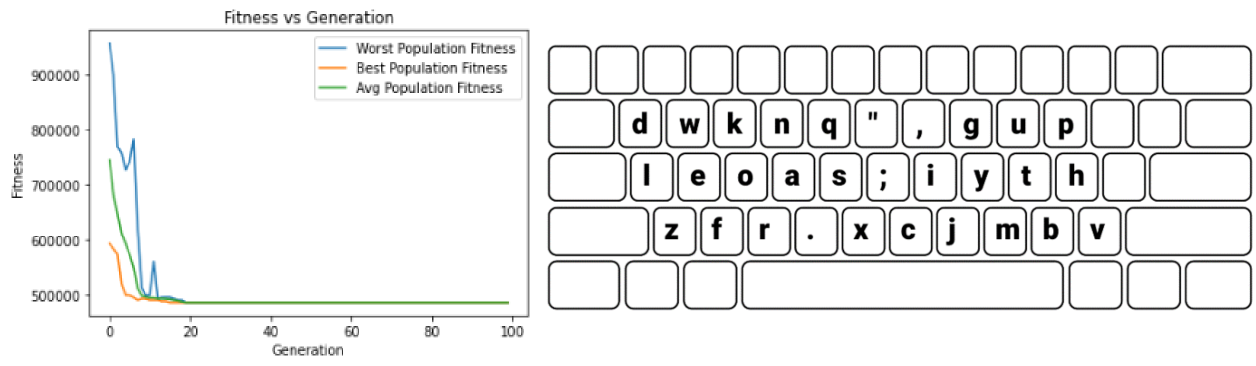
Test Case 2: Testing with a population of 100, for 100 generations, Single Point Crossover, and a crossover rate of 0.9. The results obtained (Graph showing change of population Fitness, and optimal keyboard), can be seen in Figure 16.



```
Time taken to execute:  8158.11  s
```

Figure 16: Results of Test Case 2

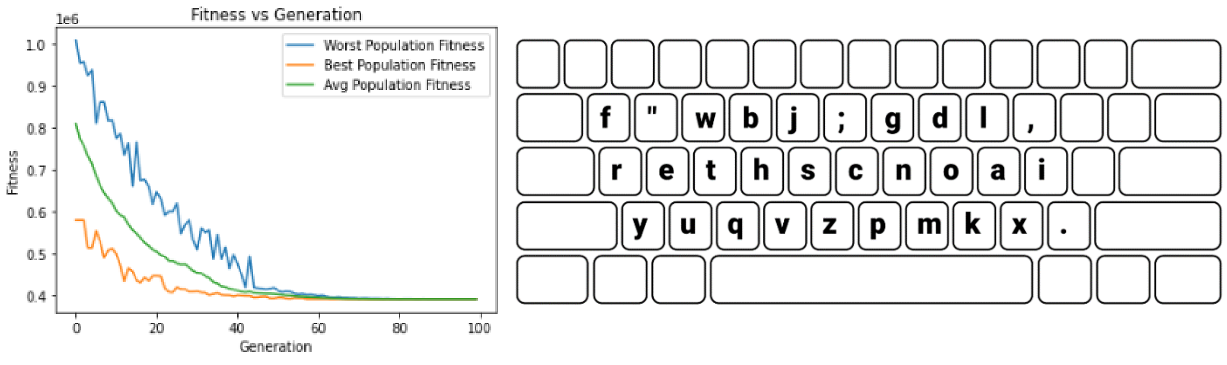
Test Case 3: Testing with a population of 100, for 100 generations, Single Point Crossover, and a crossover rate of 0.4. The results obtained (Graph showing change of population Fitness, and optimal keyboard), can be seen in Figure 17.



```
Time taken to execute:  8564.83  s
```

Figure 17: Results of Test Case 3

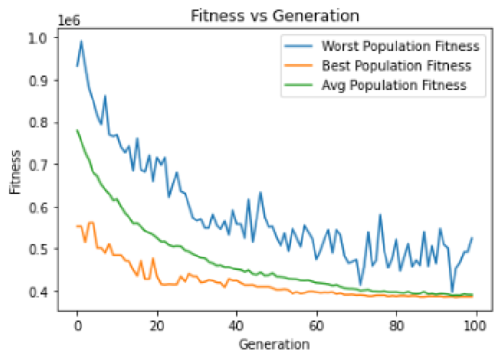
Test Case 4: Testing with a population of 100, for 100 generations, Double Point Crossover, and a crossover rate of 0.9. The results obtained (Graph showing change of population Fitness, and optimal keyboard), can be seen in Figure 18.

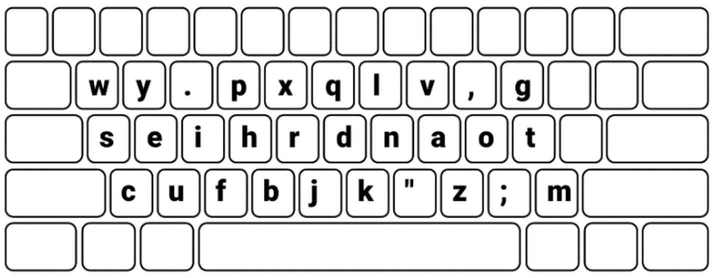


```
Time taken to execute:  8389.09  s
```

Figure 18: Results of Test Case 4

Test Case 5: Testing with a population of 100, for 100 generations, Double Point Crossover, and a crossover rate of 0.9, Swap Mutation with mutation rate of 0.1. The results obtained (Graph showing change of population Fitness, and optimal keyboard), can be seen in Figure 19.

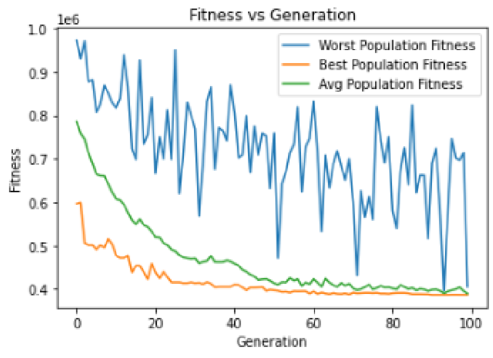


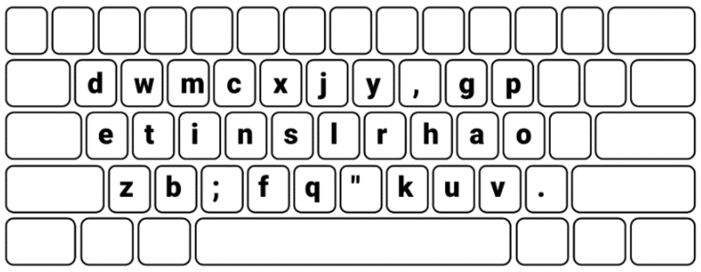


```
Time taken to execute:  8445.59  s
```

Figure 19: Results of Test Case 5

Test Case 6: Testing with a population of 100, for 100 generations, Double Point Crossover, and a crossover rate of 0.9, Scramble Mutation with mutation rate of 0.1. The results obtained (Graph showing change of population Fitness, and optimal keyboard), can be seen in Figure 20.

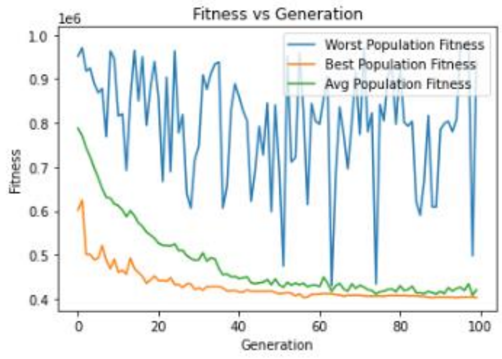


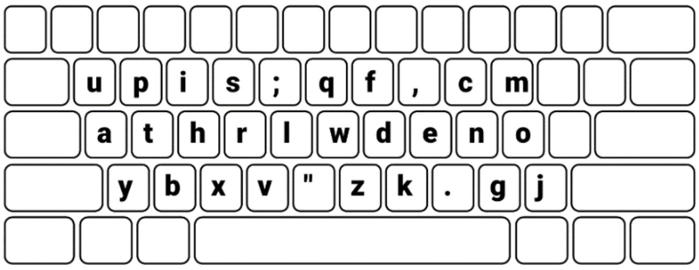


```
Time taken to execute:  8482.68  s
```

Figure 20: Results of Test Case 6

Test Case 7: Testing with a population of 100, for 100 generations, Double Point Crossover, and a crossover rate of 0.9, Inversion Mutation with mutation rate of 0.1. The results obtained (Graph showing change of population Fitness, and optimal keyboard), can be seen in Figure 21.

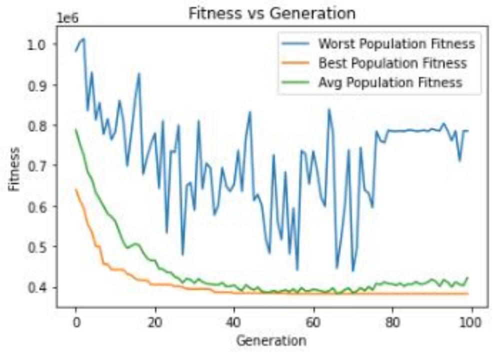


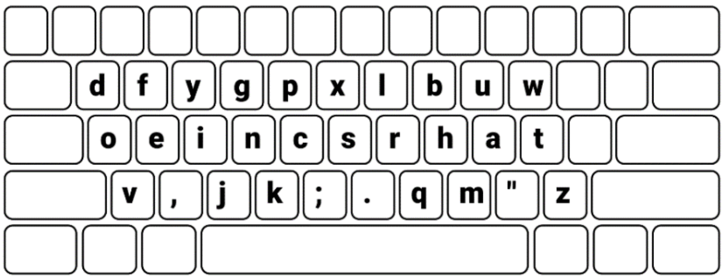


```
Time taken to execute:  8511.34  s
```

Figure 21: Results of Test Case 7

Test Case 8: Testing with a population of 100, for 100 generations, Double Point Crossover, and a crossover rate of 0.9, Scramble Mutation with mutation rate of 0.1, and elitism rate of 0.1. The results obtained (Graph showing change of population Fitness, and optimal keyboard), can be seen in Figure 22.

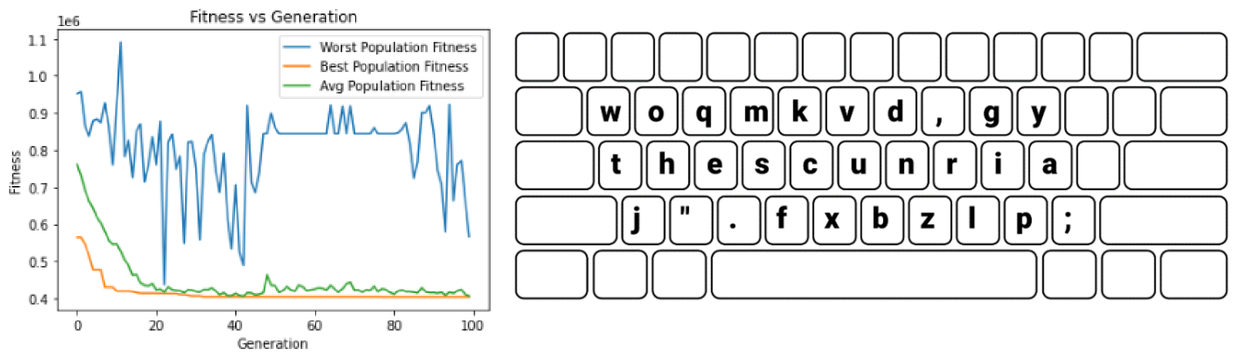




```
Time taken to execute:  8178.45  s
```

Figure 22: Results of Test Case 8

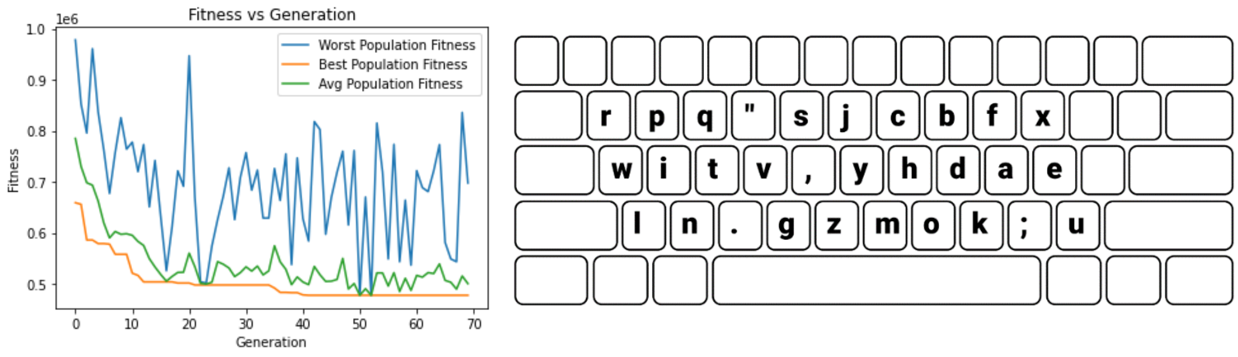
Test Case 9: Testing with a population of 100, for 100 generations, Double Point Crossover, and a crossover rate of 0.9, Inversion Mutation with mutation rate of 0.1, and elitism rate of 0.1. The results obtained (Graph showing change of population Fitness, and optimal keyboard), can be seen in Figure 23.



```
Time taken to execute:  7424.75  s
```

Figure 23: Results of Test Case 9

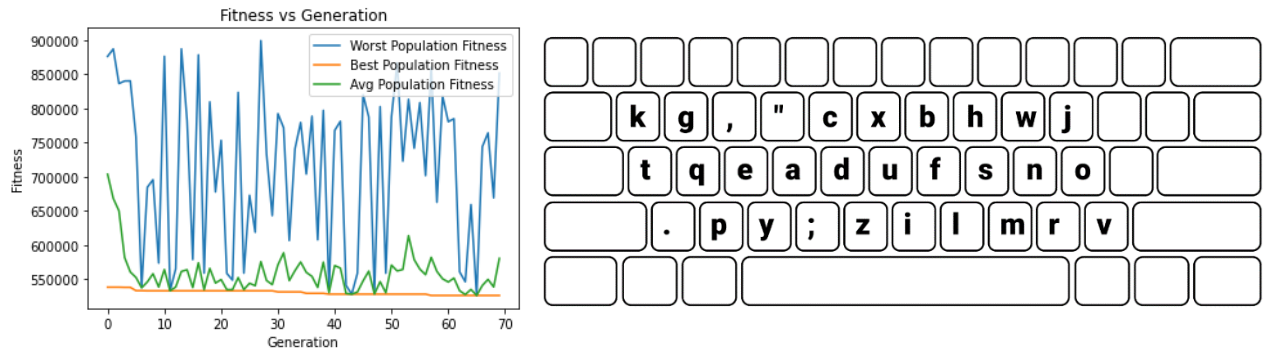
Test Case 10: Testing with a population of 15, for 70 generations, Double Point Crossover, and a crossover rate of 0.9, Scramble Mutation with mutation rate of 0.2, and elitism rate of 0.1. The results obtained (Graph showing change of population Fitness, and optimal keyboard), can be seen in Figure 24.



```
Time taken to execute:  2702.94  s
```

Figure 24: Results of Test Case 10

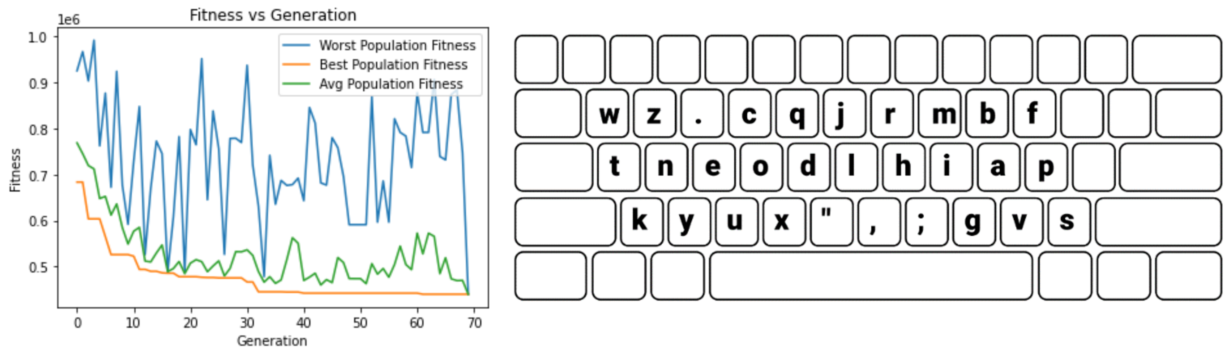
Test Case 11: Testing with a population of 15, for 70 generations, Double Point Crossover, and a crossover rate of 0.9, Inversion Mutation with mutation rate of 0.2, and elitism rate of 0.1. The results obtained (Graph showing change of population Fitness, and optimal keyboard), can be seen in Figure 25.



```
Time taken to execute:  2630.14  s
```

Figure 25: Results of Test Case 11

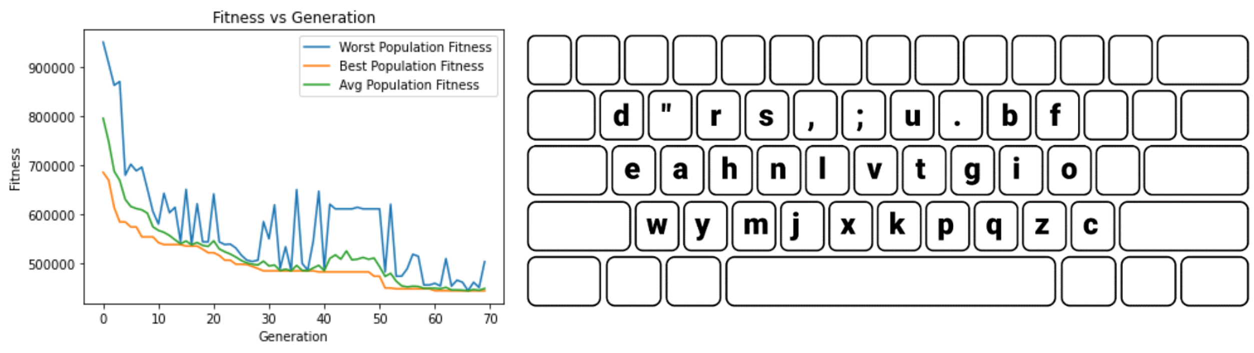
Test Case 12: Testing with a population of 15, for 70 generations, Double Point Crossover, and a crossover rate of 0.9, Inversion Mutation with mutation rate of 0.2, and elitism rate of 0.3. The results obtained (Graph showing change of population Fitness, and optimal keyboard), can be seen in Figure 26.



```
Time taken to execute:  2209.13  s
```

Figure 26: Results of Test Case 12

Test Case 13: Testing with a population of 15, for 70 generations, Double Point Crossover, and a crossover rate of 0.9, Swap Mutation with mutation rate of 0.2, and elitism rate of 0.3. The results obtained (Graph showing change of population Fitness, and optimal keyboard), can be seen in Figure 27.



```
Time taken to execute:  2179.12  s
```

Figure 27: Results of Test Case 13

Results Comparison

The following comparative table illustrates the Test cases evaluation statistics:

Keyboard Name	Keyboard Fitness	Keyboard Fitness/Character in dataset
QWERTY Keyboard	815030.085	0.7537460117672518
AZERTY Keyboard	871862.803	0.8063053409488156
Test Case 1 Best Keyboard	597991.958	0.553027503777839
Test Case 2 Best Keyboard	388875.802	0.35963529472693206
Test Case 3 Best Keyboard	485848.954	0.44931680208932534
Test Case 4 Best Keyboard	390769.78	0.36138685996378456
Test Case 5 Best Keyboard	385948.93	0.3569285012753097
Test Case 6 Best Keyboard	386663.442	0.35758928739875667
Test Case 7 Best Keyboard	403018.34	0.3727144212646559
Test Case 8 Best Keyboard	383112.911	0.35430572936800503
Test Case 9 Best Keyboard	404214.28	0.3738204356583613
Test Case 10 Best Keyboard	477716.969	0.4417962806088193
Test Case 11 Best Keyboard	526076.994	0.48651999896421544
Test Case 12 Best Keyboard	439844.839	0.40677184719219167
Test Case 13 Best Keyboard	443754.491	0.41038752305082926

Comparison of Results Discussion:

The cases outlined in the evaluation section, were based on , the following hypothesis (expected outcome);"The larger, the population size, and generation size, the longer it will take for the algorithm to converge, thus, giving the Genetic Algorithm more time to find the optimal solution. A high Crossover rate would also guarantee the discovery of the perfect keyboard, as fit parents would continue to breed more fitter children. A low Mutation rate, would provide the required effort to safeguard against any local minima, and would also account for an element of variation in the children formed. A low Elitism rate would guarantee that the top elites of the current generation will be inherited by the next generation, and thus, ensuring, that the algorithm will converge at some point, whilst also allowing the Fittest parents to be transported to the next generation. Furthermore, the larger the population and generation sizes, the more time required by the algorithm to completely execute the function. Moreover, an optimal keyboard, would consist mostly of the vowel keys being strategically placed in the position where the fingers normally rest. "

As can be seen from the above test cases, this hypothesis parameters were duly satisfied. Nevertheless, there were some surprising results as shown hereunder.

In test case 1, the Genetic algorithm, was tested with a population of 10, for 100 generations, and as expected, the Genetic Algorithm's Best Population Fitness converged quite quickly (around the 20th generation), as could be seen in the graph of Figure 15. Notwithstanding, test case 1's best keyboard had a Fitness/character of 0.55, an improvement, could already be seen, when compared to the 0.75 and 0.80 Fitness/character values of the QWERTY and AZERTY keyboards respectively. It was also interesting to note, that in the position where fingers normally rest, there were 3 vowels of test case 1's best keyboard.

Furthermore, in test case 2, when the population size was increased to 100, whilst keeping all the other values constant to test case 1, this resulted in the Genetic Algorithm's Best Population Fitness to converge slower (around the 60th generation), when compared to test case 1. In the position where fingers normally rest, there were 4 vowels of test case 2's best keyboard. Visualisations can be observed in Figure 16. Moreover, test case 2's best keyboard had a Fitness/character of 0.359, an improvement of 0.194, when compared to the 0.55 Fitness/character value of test case 1's best keyboard. Test case 2, took a larger amount of time to execute, when compared to test case 1 due to the increase in the population size.

In test case 3, when the crossover rate was reduced to 0.4, whilst keeping, all the other values constant to test case 2, this resulted in the Genetic Algorithm's Best Population Fitness to converge quicker (around the 20th generation), when compared to test case 2. For test case 3 best keyboard, in the position where fingers normally rest, there were 4 vowels. These are clearly exhibited in Figure 17. Moreover, it was also quite interesting to note, that test case 3's best keyboard had a Fitness/character of 0.44. This value was less than test case 1's best keyboard, however, it was also a higher value when compared to test case 2's best keyboard Fitness/character. Thus, one could conclude, that a high crossover rate would produce better results than a small crossover rate. Furthermore, it was also noted that the size of the population is more important than the crossover rate, with regards, to the best keyboard's Fitness.

With respect to test case 4, the crossover method utilised was changed to Double point Crossover, whilst keeping, all the other values constant to test case 2, culminating in result showing that the Genetic Algorithm's Best Population Fitness converges quicker (around the 40th generation), when compared to test case 2. For test case 4 best keyboard, in the position where fingers normally rest, there were 4 vowels. Furthermore, visualisations can be observed in Figure 18. It is worth pointing out that, test case 4's best keyboard had a Fitness/character of 0.36. This value fell in the same range as test case 2's best keyboard Fitness/character.

Consequently, in test case 5, when swap mutation with a mutation rate of 0.1 was added, whilst maintaining constant all the other values to test case 4, this resulted in the Genetic Algorithm's Best Population Fitness to converge slower (around the 90th generation), when compared to test case 4. The finger resting position for test case 5 best keyboard, again focus on placing them where there were 4 vowels. Visualisations can be observed in Figure 19. Moreover, it was also quite interesting to observe, that test case 5's best keyboard had a Fitness/character of 0.356. This value was less than test case 2's best keyboard Fitness/character. Thus, one could note the effect a degree of mutation has on the best keyboard's Fitness, and the rate at which the GA would converge.

In test case 6, the mutation method utilised was changed to Scramble Mutation. The result was that the Genetic Algorithm performed similarly to test case 5, when all the other values were constant to the ones of test case 5. The only difference between the two test cases i.e., 5 and 6, basically was that the Worst Population Fitness of test Case 6 (Figure 20), fluctuated more when compared to the Worst Population Fitness of test Case 5 (Figure 19).

Similarly, in test case 7, the mutation method utilised was changed to Inversion Mutation. Once again on keeping all the other values constant to test cases 5 and 6, the final product showed that the Genetic Algorithm performed similarly to test cases 5 and 6, with the exception that The Worst Population Fitness (Figure 21), fluctuated the most, when compared to the other mentioned cases. The findings determine that there were 3 vowels in test case 7 best keyboard as regards the fingers resting position. Test case 7's best keyboard had a Fitness/character of 0.372, quite a downgrade, upon comparing this to previous test cases, which utilised the other mutation methods, whilst retaining all other parameters constant. The results clearly indicate that, in the context of having a large population size and number of generations, Inversion mutation, will not provide, the most optimal keyboard.

Consequently, in test case 8, elitism rate of 0.1 was added, whilst keeping all the other values constant to that of test case 6. The outcome was that the Genetic Algorithm performed similarly to test case 6, however, resulting with best keyboard Fitness/character of 0.354. Remarkably, this represents a lower best keyboard Fitness/character, by a small fraction, than that of test case 6. These findings are reflected in Figure 22. An interesting observation upon comparing visualisations of test cases 8 and 6, transpired, that with the inclusion of elitism, the Best Population Fitness of the graph (Figure 22) which included elitism, converged quicker, however it had a much smoother decline, when compared to the graph in Figure 20.

Moreover, one can also note, that test case 9 gave similar results to test case 8, however, as Inversion mutation, was introduced here, which failed to come up with a better optimal l Keyboard than that of test case 8. The visualisations of test case 9, can be observed, in Figure 23.

Test cases 10 till 13, were inspired from the study presented in [7], and were tested to check whether an optimal keyboard, which had a Fitness/character, like test cases 2 till 9, exists, without incurring the time complexity of running the algorithm, with a population size of 100, for 100 generations.

In test case 10, the Genetic algorithm, was tested with a population of 15, for 70 generations and with a mutation rate of 0.2, whilst keeping all the other values constant to test case 8. The Genetic Algorithm converged quite slowly (around the 40th generation), as could be seen in the graph of Figure 24. Notwithstanding, test case 10's best keyboard had a Fitness/character of 0.44, an improvement, could already be seen, when compared to the 0.75 and 0.80 Fitness/character values of the QWERTY and AZERTY keyboards respectively. Unfortunately, such statistic had a difference of 0.1, when compared to the Fitness/character, of the other cases. Test case 10 took relatively, less time to compute when compared to the previous test cases. Once again in this test case fingers position normally rest, in the vicinity of 3 vowels.

Moreover, one can also note, that test case 11 gave similar results to test case 10. The visualisations of test case 11, can be observed, in Figure 25. The change between the two test cases i.e., 10 and 11, pertained to the change in the mutation method, which didn't result in any significant difference.

In test case 12, the Genetic algorithm, was tested with an elitism rate of 0.3, whilst keeping all the other values constant to that of test case 10. The Genetic Algorithm performed similarly to test case 10, however it had a

Fitness/character of 0.40, which one notes was less than one observed in test case 10. The Best Population Fitness of test case 12, as observed in Figure 26, had a much smoother decline, than the one observed in Figure 24, such difference pertains to the increase in the elitism rate, for smaller GA's which have a smaller population. For test case 12 best keyboard, the fingers' resting position was where there were 4 vowels.

Conclusively, one can also note, that test case 13 gave similar results to test case 12. The visualisations of test case 13, can be observed, in Figure 27. The change between the two test cases pertained to the change in the mutation method and didn't result in any significant difference.

Lastly, a test, was conducted on the most optimal keyboard derived from the 13 test cases, which was the keyboard in test case 8, (keyboard in Figure 22), to determine whether this would still rank as the optimal keyboard, when tested with the two most frequently used keyboard, with a different dataset. The dataset which the keyboards were tested with, was obtained from [8]. The following results were obtained:

Keyboard Name	Keyboard Fitness	Keyboard Fitness/Character in dataset
QWERTY Keyboard	381725.159	0.7626236339807648
AZERTY Keyboard	408136.181	0.8153884808867188
Test Case 8 Best Keyboard (Most optimal Keyboard)	180436.382	0.3604820015103628

Thus, one can easily conclude, that the optimal keyboard also produces an optimal Fitness/character, when other datasets are utilised. In addition, the Genetic Algorithm is not overfitting, as the Fitness/character of the optimal keyboard when testing it with the both the first dataset and the second dataset, are similar.

Conclusion:

As a result of this study, various optimal keyboard configurations have been constructed, and all configurations, have been proven to be more optimal when compared to the QWERTY and AZERTY keyboard layouts. Nevertheless, the purpose of this study of establishing whether the creation of an improved keyboard on the existing two ones, is a realistic and attainable target. The scope of such study was achieved with the creation of the most optimal keyboard configuration exhibited at Test Case 8. Moreover, throughout this study, various patterns were observed, and it could be said, that there isn't truly one Perfect Keyboard, but there are multiple, as there is more than one way to achieve optimality.

In summary, these interesting patterns relating to Genetic Algorithms, which were observed throughout this research, include the following:

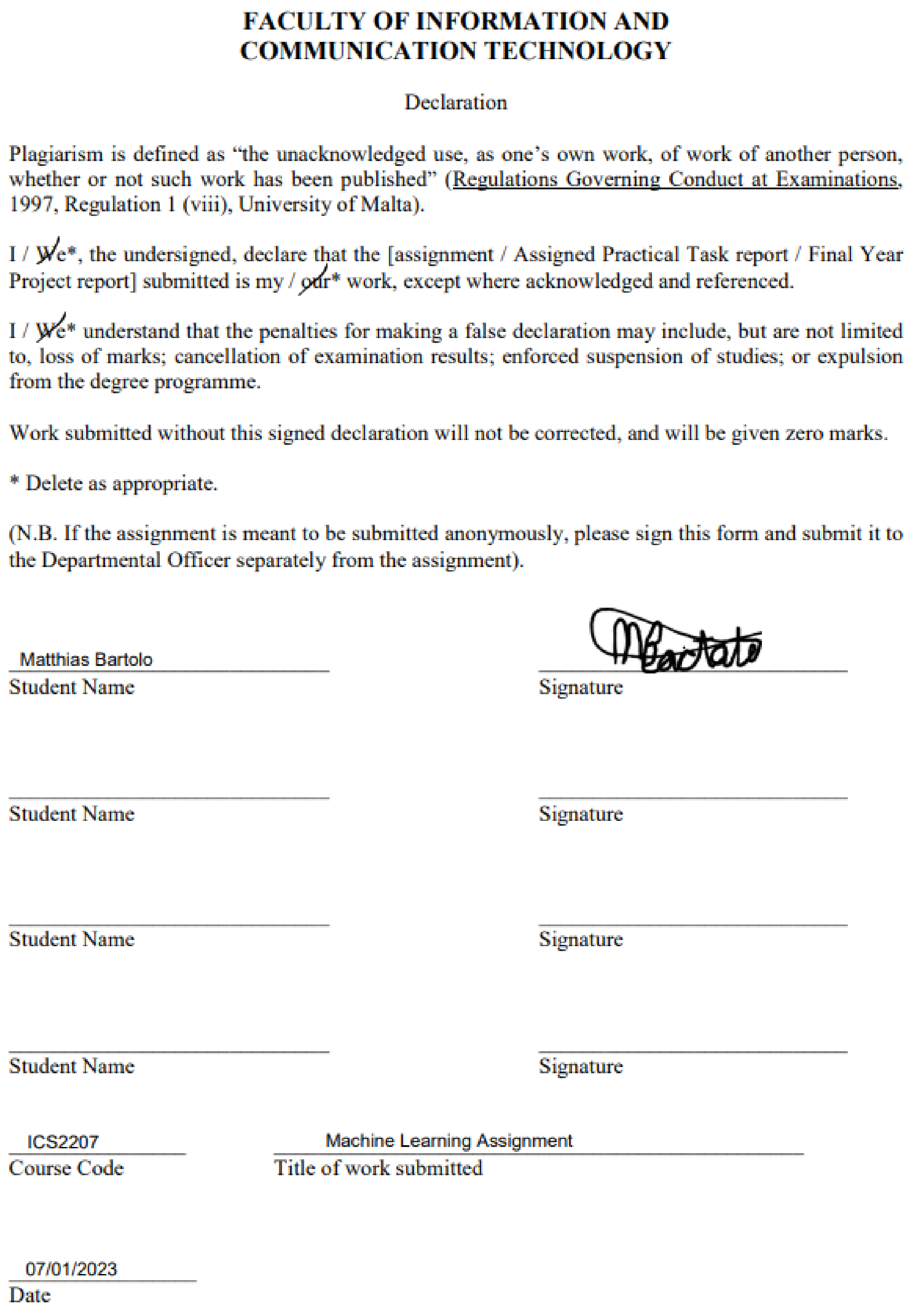
1. Two-Point Crossover usually generates better results, in comparison to Single-Point Crossover.
2. A High Crossover Rate generates better outcomes, when compared to a Low Crossover Rate, as the latter restricts the rate at which, the average Keyboard Fitness could improve. As a result, the Genetic Algorithm would converge relatively early, prohibiting the algorithm from reaching its most optimal result.
3. Mutation is a staple mechanism, which prevents the Genetic Algorithm from getting caught in any Local Minima.
4. A Low Mutation Rate, typically in the range of 0.05 and 0.35, generates preferable results, as it enables the Genetic Algorithm, to reach the Global Minima. Consequently, a High Mutation Rate is not beneficial, as it would increase the probability of searching in a larger search space, and thus would inhibit the Genetic Algorithm to converge.
5. Elitism is also a vital procedure, which enables the best Keyboard Fitness to either decrease or remain static, across the span of multiple generations. Nevertheless, a High Elitism Rate, will enable the Genetic Algorithm to converge quite early on, disallowing the algorithm from reaching its most ideal result.

Conclusively, the keyboard constructed during this study, pertained to a keyboard, which one uses, when typing words from the English language. One may find that attempting to use this keyboard to type another language, may not lead to the same successful result and solution. Paving the way to the conduction of other research, which explore potential Perfect Keyboards, for different languages, along with the challenge of discovering a better keyboard configuration, which surpasses the Perfect Keyboard discovered in this study.

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
If partial, explain what has been done	

<u>Plagiarism Declaration Form:</u>



FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

Plagiarism is defined as "the unacknowledged use, as one's own work, of work of another person, whether or not such work has been published" (<u>Regulations Governing Conduct at Examinations</u>, 1997, Regulation 1 (viii), 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 making a false declaration may include, but are not limited to, 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).

Matthias Bartolo	[illegible]
Student Name	Signature
Student Name	Signature
Student Name	Signature
Student Name	Signature

ICS2207	Machine Learning Assignment
Course Code	Title of work submitted

07/01/2023
Date

References:

[1] - V. Mallawaarachchi "Introduction to Genetic Algorithms — Including Example Code" 2017 [Online]. Available: https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3. [Accessed: 22- Dec- 2022]

[2] - adumb, Using AI to Create the Perfect Keyboard 2022 [Online video]. Available: https://www.youtube.com/watch?v=EOaPb9wrgDY . [Accessed: 22- Dec- 2022]

[3] - Project Gutenberg, "Ben-Hur: A tale of the Christ by Lew Wallace" [Online]. Available: https://www.gutenberg.org/ebooks/2145. [Accessed: 22- Dec- 2022]

[4] - Tim's Printable, Free Printable Blank Keyboard Template Printable [Online image]. Available: https://timvandevall.com/free-blank-keyboard-template-printable/ . [Accessed: 22- Dec- 2022]

[5] - C. Robertson, "Roboto", Google Fonts. [Online font]. Available: https://fonts.google.com/specimen/Roboto. [Accessed: 22- Dec- 2022]

[6] - tutotrialspoint, "Genetic Algorithms - Mutation" [Online]. Available: https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_mutation.htm. [Accessed: 22- Dec- 2022]

[7] - R.L. Haupt, "Optimum population size and mutation rate for a simple ... - IEEE xplore." 2002 [Online]. Available: https://ieeexplore.ieee.org/document/875398 . [Accessed: 24-Dec-2022].

[8] - Project Gutenberg, "The Story of Malta by Maturin M. Ballou" [Online]. Available: https://www.gutenberg.org/ebooks/34036 . [Accessed: 24- Dec- 2022]