Machine Learning Project: Keyboard Optimisation Genetic Algorithm Report

Name: Nicholas Falzon
ID: 19603H

Table of Contents

Statement of Completion

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

Introduction

As described in the project specifications, I have implemented a genetic algorithm capable of designing an optimised keyboard which reduces the distance that a user moves their fingers in order to type.

Throughout this report I will explain each function which I have implemented in python, all of which execute a crucial part of the genetic algorithm. In each section I will display the section of the code which I am referring to at that moment, however I have also included the .ipynb file which I worked on in the submission folder.

Design Decisions

Libraries Used

```
import nltk
from nltk.tokenize import word_tokenize
from operator import itemgetter
import random
import math
import matplotlib.pyplot as plt
import numpy as np
from numpy.random import choice
import pandas as pd
```

Choice of Corpus

I chose to obtain a corpus from the link provided in the project specifications and chose to use My Life — Volume 1 by Richard Wagner [1] which I included in the submission folder.

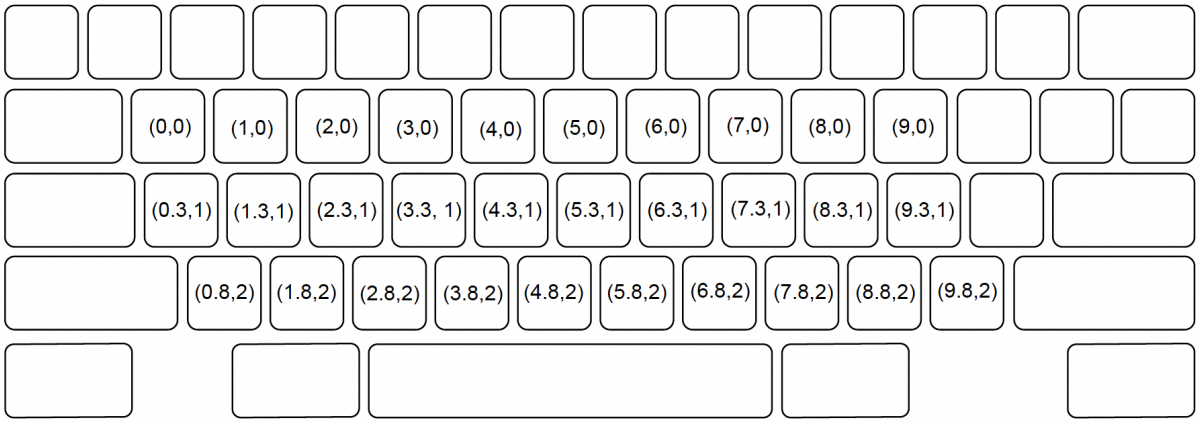
Regarding pre processing the data, I tokenized the text and case-folded all the characters to lowercase using commands provided by the nltk library.

```
txt = open("text.txt", "r")
tokens = [t.lower() for t in word_tokenize(txt.read())] #splitting text
into tokens and case-folding to lowercase
```

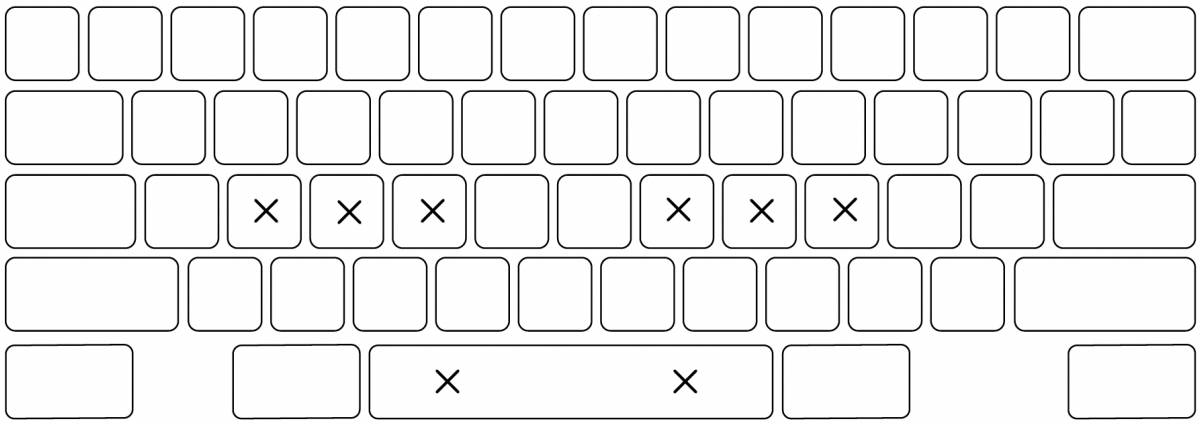
Chromosome Design

I implemented the chromosomes (keyboards) through the use of python dictionaries. I found these particularly useful as they allowed me to store each keyboard key in a pair along with its corresponding location. I determined the location of each key by assigning it with a coordinate to try to represent the positions of the keys on a real life keyboard.

The coordinates which I used to represent the positions of each real life key position can be seen below:



I chose to assume that the user will use 8 fingers to type since this mimics how I type as I don't use any of my little fingers when typing. The assumed starting positions of these fingers can be seen below:



I am assuming that the two thumbs will be positioned on the spacebar keys and therefore in reality only 6 fingers are being used, similar to the scenario in the video provided in the project description.

Generate Initial Population Function

This function will generate the first generation of keyboards. It will receive as a parameter the size of the population i.e the number of keyboards to be present in each generation and return a number of keyboards accordingly. The initial population of keyboards will be generated randomly by picking a random spot for each character to be placed in.

```
def generate_initial_population(population_size): # Function which
generates an n number of keyboards with key positions randomised
    key_placements = []
    for _ in range(population_size):
        key_placement = {}
        chars = "qazwsxedcrfvtgbyhnujmik,ol.p;?"

        placements = [(0,0), (0.4,1), (0.8,2),
                      (1,0), (1.4,1), (1.8,2),
                      (2,0), (2.4,1), (2.8,2),
                      (3,0), (3.4,1), (3.8,2),
                      (4,0), (4.4,1), (4.8,2),
                      (5,0), (5.4,1), (5.8,2),
                      (6,0), (6.4,1), (6.8,2),
                      (7,0), (7.4,1), (7.8,2),
                      (8,0), (8.4,1), (8.8,2),
                      (9,0), (9.4,1), (9.8,2)]

        for char in chars:
            placement = random.choice(placements)
            placements.remove(placement) # Removing position so that it
cannot be chosen again
            key_placement[char] = placement
        key_placements.append(key_placement)
    return key_placements
```

Fitness Function

The calculate_fitness function will calculate the distance that a user must move his fingers in order to type every character in the text corpus for each keyboard layout. The fitness value will determine how efficient a keyboard is in writing the text corpus. The lower the value, the more efficient a keyboard is.

Each finger is responsible for typing a set of characters which consist of those characters that are adjacent to it. The positions each finger is responsible for is specified inside the various if or elif statements. Some fingers are responsible for more characters than others since there are no other fingers closer to those characters, such as those fingers positioned at (1.3, 1), (3.3, 1), (6.3, 1) and (8.3, 1). If a finger must be moved to access an adjacent character, the position of the finger is changed to the position of the character moved towards.

I used commands such as .isascii() and .isalpha() in the calculate_fitness function in order to make sure that only the thirty characters which were specified in the keyboard are checked for distance.

After each keyboard's distance is calculated, a dictionary element is created with the key being the index of the keyboard within the list that it is stored in and the value being the keyboard's corresponding fitness value.

```
def calculate_fitness(keyboard_list, text): # Function that calculates the
total distance required to type a give text for each keyboard layout
provided
    keyboard_distances = {}
    for i, keyboard in enumerate(keyboard_list):
        # Defining the key which each finger will originally be placed on
        f1 = list(keyboard.keys()) [list(keyboard.values()).index((1.4,
1))]
        f2 = list(keyboard.keys()) [list(keyboard.values()).index((2.4,
1))]
        f3 = list(keyboard.keys()) [list(keyboard.values()).index((3.4,
1))]
        f4 = list(keyboard.keys()) [list(keyboard.values()).index((6.4,
1))]
        f5 = list(keyboard.keys()) [list(keyboard.values()).index((7.4,
1))]
        f6 = list(keyboard.keys()) [list(keyboard.values()).index((8.4,
1))]
```

```
        d = 0

        chars = [i for ele in tokens for i in ele]

        for char in chars:
            if (char.isascii() and char.isalpha()) or char == ';' or char
== ',' or char == '.' or char == '?': # isascii() makes sure that no
non-english letters are considered while isalpha() makes sure that no
numerical values are considered
                key_location = keyboard[char]
                # Defining the positions which each finger is responsible
for typing
                if key_location == (0, 0) or key_location == (0.3, 1) or
key_location == (0.8, 2) or key_location == (1, 0) or key_location == (1.3,
1) or key_location == (1.8, 2):
                    d += math.dist(key_location, keyboard[f1])
                    f1 = char
                elif key_location == (2, 0) or key_location == (2.3, 1) or
key_location == (2.8, 2):
                    d += math.dist(key_location, keyboard[f2])
                    f2 = char
                elif key_location == (3, 0) or key_location == (3.3, 1) or
key_location == (3.8, 2) or key_location == (4, 0) or key_location == (4.3,
1) or key_location == (4.8, 2):
                    d += math.dist(key_location, keyboard[f3])
                    f3 = char
                elif key_location == (5, 0) or key_location == (5.3, 1) or
key_location == (5.8, 2) or key_location == (6, 0) or key_location == (6.3,
1) or key_location == (6.8, 2):
                    d += math.dist(key_location, keyboard[f4])
                    f4 = char
                elif key_location == (7, 0) or key_location == (7.3, 1) or
key_location == (7.8, 2):
                    d += math.dist(key_location, keyboard[f5])
                    f5 = char
                elif key_location == (8, 0) or key_location == (8.3, 1) or
key_location == (8.8, 2) or key_location == (9, 0) or key_location == (9.3,
1) or key_location == (9.8, 2):
                    d += math.dist(key_location, keyboard[f6])
                    f6 = char
            else: pass # If character is not one of those we are checking
for, ignore it
```

```
        keyboard_distances[i] = d

    return keyboard_distances
```

Elitism

Elitism was implemented through identifying the two fittest chromosomes of each generation and immediately passing those chromosomes into the next generation untouched. This was done by sorting the fitness values in ascending order and passing through the fittest keyboards (lowest distance). The elite keyboards were also made available to be selected in each generation through the selection function.

The implementation of elitism assures that the elite keyboards are kept alive which is crucial since the most optimal keyboard solution will likely build off of these elite keyboards.

```
def elitism(population, keyboard_distances):
    best_keyboards = []

    values = sorted(keyboard_distances.values()) # Sorting fitness values
in ascending order

    index1 =
list(keyboard_distances.keys())[list(keyboard_distances.values()).index(val
ues[0])] # Obtaining the index of the keyboard with the best (lowest)
fitness value

    # Using a for loop to make sure that the second keyboard returned is
not the exact same as the first
    for value in values[1:]:
        if value > values[0]:
            index2 =
list(keyboard_distances.keys())[list(keyboard_distances.values()).index(val
ue)]
            break

    best_keyboards.append(population[index1])
    best_keyboards.append(population[index2])

    return best_keyboards
```

Selection Function

The selection function will take as input the current population and the corresponding fitness of each chromosome. It will firstly create a list containing the population of keyboards which will be sorted by fitness in ascending order, list[0] being the most fit solution and list[last] being the least fit. From there, the first half of this list will be extracted. Only the keyboards present in this first half of the list will be able to be picked to be crossed over to the next generation. The reason I decided to do this is to try to increase the chance that the offset keyboards will be ones created from parent keyboards with at least a decent fitness value.

One possible problem which may be encountered when using this approach, is that using this method with a small population size could lead to premature convergence, which is a problem that arises when the genes of some high rated individuals quickly start dominating the population, constraining it to converge early and making it suboptimal. I will discuss this in further detail in the upcoming sections.

After obtaining the first half of the list, I implemented an algorithm to select which of the top 50% of keyboards are to be selected. In the end I decided to opt for **rank selection**.

Rank selection works by assigning a “rank” to each chromosome which can be selected. The rank assigned to each chromosome will determine the probability it has to be selected. The chromosome with the worst fitness value will be given the worst rank, which is 1, second worst will be given rank 2, and this process is repeated until the best chromosome is given rank n, where n is the number of chromosomes available for selection. Each chromosome’s rank will then be divided by the Gauss Summation of n in order to obtain the probability of each chromosome to be selected. Two keyboards are then selected based on these probabilities.

I decided to use rank selection since I have noted from certain studies [2] that using it leads to having a less chance of the population becoming prematurely converged, which is something I suspected was happening when I was running the evolution with a small population with a different selection algorithm. The implementation of rank selection definitely helped in this regard as the population did not converge as quickly as it did before, however very similar or identical chromosomes can sometimes still be seen early on in the evolution when using a small population. The obvious fix to this would be to run the entire evolution with a greater population size, which I have done and will be discussing later on.

```
def select_keyboards(population, keyboard_distances): # Selection function which chooses 2 keyboards from the top 50% of keyboards of the given population
    selected_keyboards = []

    # Sorting the dictionary to identify the best keyboards
    keys = list(keyboard_distances.keys())
    values = list(keyboard_distances.values())
    sorted_value_index = np.argsort(values)
    sorted_dict = {keys[i]: values[i] for i in sorted_value_index}

    keyboards = []

    # Creating a list of keyboards which will be sorted from best fitness to worst
    for i in sorted_dict.keys():
        keyboards.append(population[i])

    length = len(sorted_dict)
    middle_index = length//2

    # Obtaining the first half of the list which will be the top 50% of keyboards with the best fitness score
    first_half = keyboards[:middle_index]

    # Rank Based Selection:

    n = int(len(population) / 2)

    rank_sum = n * (n + 1) / 2 # Gauss Summation

    probability = []

    # Calculating the probability of each chromosome to be selected
    for i in list(range(n,0,-1)): # For loop which starts from n and decrements i by 1 until i = 1
        probability.append(i/rank_sum)

    selected_keyboards = choice(first_half, 2, replace = False, p = probability)

    return selected_keyboards
```

Two-point Crossover Function

The crossover function takes two parameters as input, those being the two parent keyboards which are to be combined, and returns a list containing the two offspring keyboards which are generated by the crossover function.

Two split points are randomly generated each time the crossover function is called, one of which is a point from the left-hand side of the keyboard while the other is from the right hand side of the keyboard. The first keyboard will be constructed from all the keys which are located to the left of the left split point of keyboard1 and all the keys to the right of the right split point of keyboard1. The remaining keys are filled with those from keyboard2 in the order in which they appear after the left split point (top to bottom, left to right, similar to that of the youtube video). The other keyboard will be generated in the same manner however the keyboards are changed, the keys to the left of the left split point of keyboard2 and those to the right of the right split point of keyboard2 are inserted into the new keyboard while the remaining are filled with those from keyboard1.

```
def crossover(parent_keyboard1, parent_keyboard2):
    offset_keyboards = []

    # Two Point Crossover

    offset_parent_keyboard1 = {}
    offset_parent_keyboard2 = {}

    placements = [(0,0), (0.4,1), (0.8,2),
                  (1,0), (1.4,1), (1.8,2),
                  (2,0), (2.4,1), (2.8,2),
                  (3,0), (3.4,1), (3.8,2),
                  (4,0), (4.4,1), (4.8,2),
                  (5,0), (5.4,1), (5.8,2),
                  (6,0), (6.4,1), (6.8,2),
                  (7,0), (7.4,1), (7.8,2),
                  (8,0), (8.4,1), (8.8,2),
                  (9,0), (9.4,1), (9.8,2)]

    # 2 splitpoints are chosen - 1 from the left side of the keyboard and
the other from the right side

    placements_left = [(0,0), (0.4,1), (0.8,2),
                       (1,0), (1.4,1), (1.8,2),
```

```
                      (2,0), (2.4,1), (2.8,2),
                      (3,0), (3.4,1), (3.8,2),
                      (4,0), (4.4,1), (4.8,2)]

    placements_right = [(5,0), (5.4,1), (5.8,2),
                        (6,0), (6.4,1), (6.8,2),
                        (7,0), (7.4,1), (7.8,2),
                        (8,0), (8.4,1), (8.8,2),
                        (9,0), (9.4,1), (9.8,2)]

    splitpoint_left = random.choice(placements_left)
    splitpoint_right = random.choice(placements_right)

    indexleft = placements_left.index(splitpoint_left)

    # All keys to the left of the left splitpoint are inserted into the
offset keyboard
    # at the position in which they were located in the parent keyboard

    for key in parent_keyboard1.keys():
        if parent_keyboard1[key] in placements_left[:indexleft]:
            offset_parent_keyboard1[key] = parent_keyboard1.get(key)

    for key in parent_keyboard2.keys():
        if parent_keyboard2[key] in placements_left[:indexleft]:
            offset_parent_keyboard2[key] = parent_keyboard2.get(key)

    indexright = placements_right.index(splitpoint_right)

    # All keys at or to the right of the right splitpoint are inserted into
the offset keyboard
    # at the position in which they were located in the parent keyboard

    for key in parent_keyboard1.keys():
        if parent_keyboard1[key] in placements_right[indexright:]:
            offset_parent_keyboard1[key] = parent_keyboard1.get(key)

    for key in parent_keyboard2.keys():
        if parent_keyboard2[key] in placements_right[indexright:]:
            offset_parent_keyboard2[key] = parent_keyboard2.get(key)

    # Defining the order in which the parent keyboards will be traversed in
order to fill in the rest of the offset keyboards
```

```
    order = placements[indexleft+1:] + placements[:indexleft+1]

    # Keys are inserted into the offset keyboard depending on the position
in which they appear after the first splitpoint
    for pos in order:
        keys = list(parent_keyboard2.keys())
        values = list(parent_keyboard2.values())

        position = values.index(pos)
        key = keys[position]

        if len(offset_parent_keyboard1.values()) == 30: break

        elif key not in offset_parent_keyboard1.keys():
            for i in placements:
                if i not in offset_parent_keyboard1.values():
                    offset_parent_keyboard1[key] = i
                    break
    offset_keyboards.append(offset_parent_keyboard1)

    for pos in order:
        keys = list(parent_keyboard1.keys())
        values = list(parent_keyboard1.values())

        position = values.index(pos)
        key = keys[position]

        if len(offset_parent_keyboard2.values()) == 30: break

        elif key not in offset_parent_keyboard2.keys():
            for i in placements:
                if i not in offset_parent_keyboard2.values():
                    offset_parent_keyboard2[key] = i
                    break
    offset_keyboards.append(offset_parent_keyboard2)

    return offset_keyboards
```

Run Evolution Function

This function makes use of all the aforementioned functions in order to run the full genetic algorithm.

```
def run_evolution():
    population = generate_initial_population(10)
    number_of_generations = 500
    low = []
    gens = []
    avg = []
    for i in range(number_of_generations):
        gens.append(i)
        print("Generation {}:".format(i+1))
        print("")
        keyboard_distances = calculate_fitness(population, tokens)
        print(keyboard_distances)
        print("")
        avg.append(sum(keyboard_distances.values()) / 10)
        print(sum(keyboard_distances.values()) / 10)
        print("")
        lowest = min(keyboard_distances.values())
        low.append(lowest)
        next_generation = elitism(population, keyboard_distances)

        for j in range(int(len(population) / 2) - 1):
            parents = select_keyboards(population, keyboard_distances)
            new_keyboards = crossover(parents[0], parents[1])
            next_generation += new_keyboards

        population = next_generation

    %matplotlib qt
    plt.figure(1)
    plt.plot(gens, low, c = "red", linewidth = 2)
    plt.title("Lowest Distance")
    plt.xlabel("Generation")
    plt.ylabel("Distance")
    plt.show()

    plt.figure(2)
    plt.plot(gens, avg, c = "red", linewidth = 2)
```

```
    plt.title("Average Distance")
    plt.xlabel("Generation")
    plt.ylabel("Distance")
    plt.show()

    return population
```

Expected Outcomes

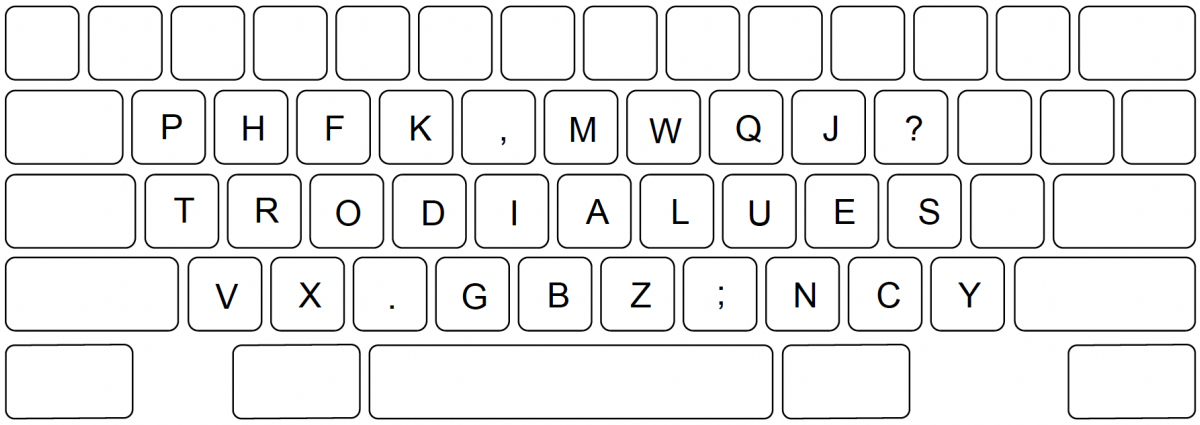
Before providing my findings, I would like to elaborate on what I expected and aimed to happen when I ran the genetic algorithm.

Naturally, I expected the average and lowest distance of the keyboards to be reduced as they passed through the generations as only the fittest keyboards would be used to create offset keyboards which are passed through to future generations. In the end, after all the generations were passed I expected that the most frequently present characters in my text corpus would be placed in the positions where the fingers are placed. This is because the user does not need to move his fingers in order to type these characters, therefore reducing distance taken. Additionally, I also expected the least frequently present characters to be placed in the positions which are furthest away from the fingers, those being (4.8, 2) and (9.8, 2).

I also expected that the evolution ran with a larger number of populations would provide the more optimised keyboard purely based on the fact that I read that rank selection works well with larger populations.

Results

After running the evolution multiple times with different parameters such as population size, using/not using elitism, and using different crossover methods, the below keyboard is the one which managed to provide the lowest fitness score:



I managed to obtain this layout from running the evolution with a population of 100 for 500 generations.

I have decided to label it the “RODLUE” keyboard as these are the keys which the fingers are placed on.

As previously mentioned, I expected the most common characters in my corpus to be placed in the positions where the user’s fingers are placed. In order to check for this, I ran the below code in order to produce the number of occurrences of each character in my corpus:

```
from collections import Counter

chars = [i for ele in tokens for i in ele]

output = Counter(chars)

print(output.most_common())
```

Which produced this output:

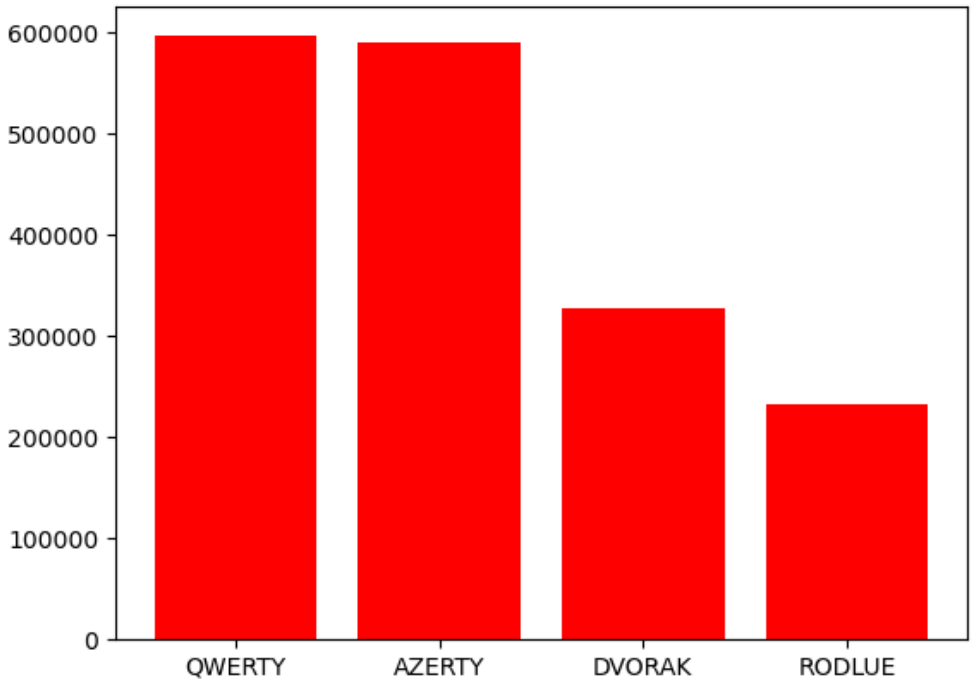
```
[('e', 128460), ('t', 93705), ('a', 79339), ('o', 76253), ('i', 75025), ('n', 69896), ('r', 62349), ('s', 62225),
('h', 59574), ('d', 43076), ('l', 38033), ('c', 31146), ('m', 30483), ('u', 28397), ('f', 26271), ('w', 21867), ('y',
20220), ('p', 19769), ('g', 18994), (',', 15019), ('b', 13680), ('v', 9745), ('.', 7038), ('k', 5258), ('x', 1954),
('’', 1343), ('-', 1184), ('q', 1181), ('j', 1151), ('z', 1013), (';', 762), ('‘', 373), ('(', 226), (')', 226),
('—', 221), ('1', 221), (':', 190), ('ä', 155), ('ö', 129), ('8', 121), ('ü', 111), ('4', 81), ('!', 81), ('3', 63),
('2', 58), ('_', 56), ('0', 46), ('?', 45), ('[', 38), (']', 38), ('5', 31), ('6', 30), ("'", 29), ('9', 28), ('`', 2
2), ('7', 21), ('*', 16), ('ï', 14), ('é', 12), ('/', 10), ('œ', 7), ('î', 3), ('è', 3), ('“', 2), ('”', 2), ('à',
2), ('$', 2), ('\ufeff', 1), ('#', 1), ('…', 1), ('û', 1), ('ù', 1), ('%', 1)]
```

The characters which ended up being placed at the positions where the fingers are placed are: r, o, d, l, u and e. If we take a look at the frequency of the mentioned characters, it can be noted that they are all among the most frequent characters. Furthermore, characters such as t, a, i and s which are also very frequent are placed in the slots which are horizontally agent to those with fingers on them, meaning they have to travel the least possible distance except for being on the character to start with.

The character I am disappointed with the position of is n since it is the 6th most frequent character however it ended up being placed in the bottom row. Additionally, the keys placed at locations (4.8, 2) and (9.8, 2) are b and y respectively, and while they are by all means not the most frequent, they're not the least frequent either which means the keyboard could have been optimised further. However I understand that a genetic algorithm is meant to provide an optimal solution and not a perfect one, and I think it managed to do that.

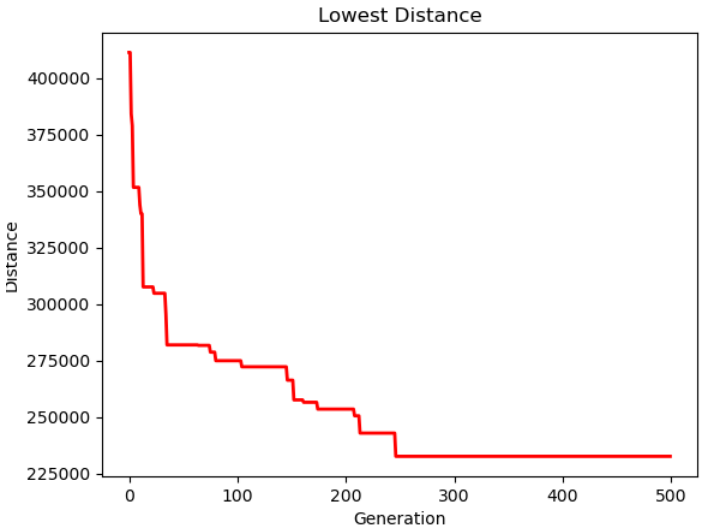
Comparison to Traditional Keyboard Layouts

When comparing the distance taken to write the entire text corpus, it is clear that the optimised keyboard is significantly more efficient than traditional keyboard layouts such as the QWERTY or AZERTY layouts, performing 61% better than the QWERTY layout and 60.5% better than the AZERTY layout. The optimised keyboard also managed to outperform the DVORAK layout, not as significantly as the others however still 29% better. The QWERTY layout got a distance of 596276.74, the AZERTY 590388.20, the DVORAK 326911.15, and the RODLUE 232602.14, as can be seen below:

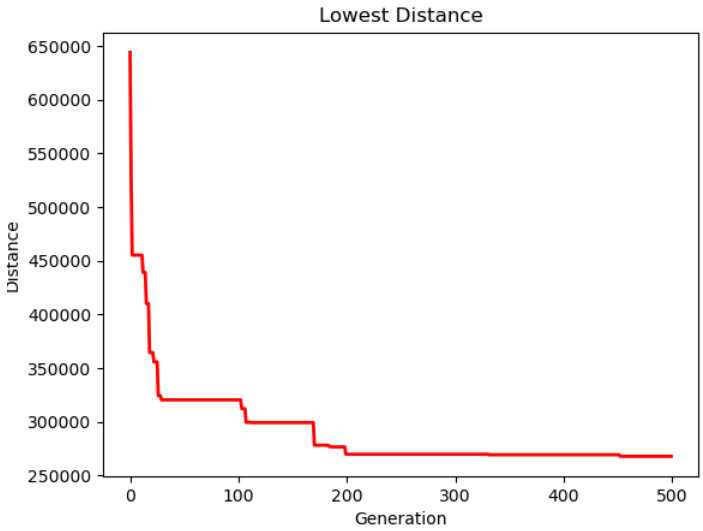


The Effect of Population

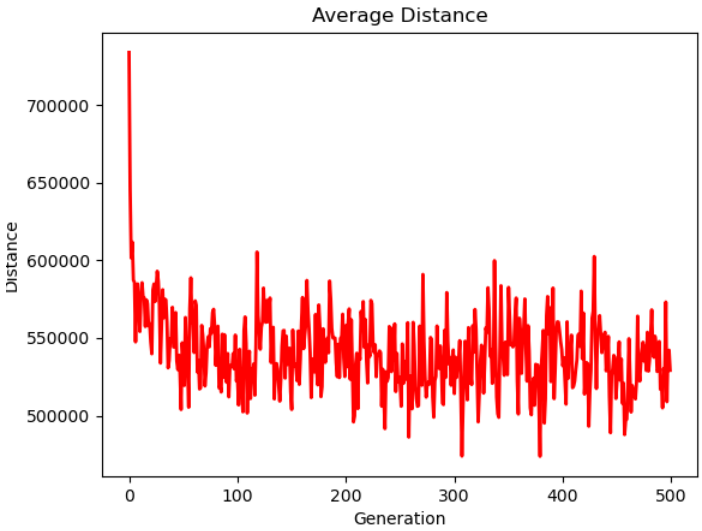
Prior to running the evolution with a population of 100, I ran it with a population of 10. Below you can see the average distance and lowest distance of each as they passed through the generations.



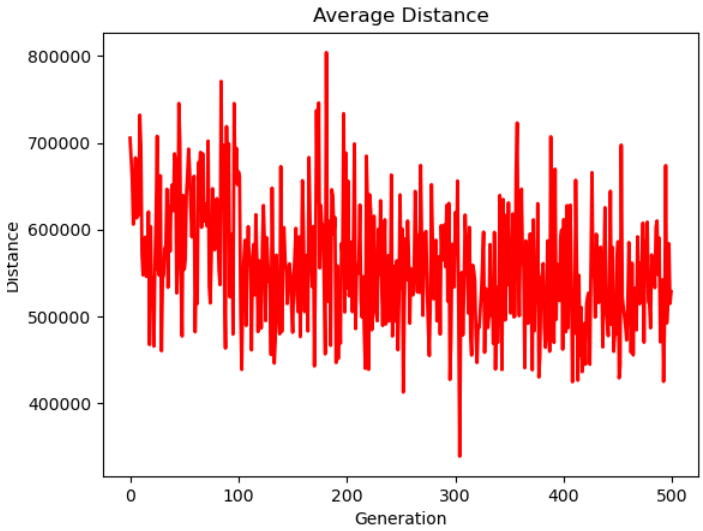
Population of 100



Population of 10



Population of 100



Population of 10

Firstly, I must point out the most important point which is that the evolution which ran with a population of 100 managed to produce a better final keyboard than that which the one with 10 produced. The final keyboard of the 100 population has a fitness of 232602.14 while the one with 10 has a fitness of 267930.18.

There are a multitude of reasons as to why this happened. First of all, producing ten times more of the amount in the initial population (the random keyboards) will result in a high probability that the lowest fitness value after the first generation will be lesser than that of the evolution ran with a population of 10. This can be visually seen in the graphs of the lowest distance, where the population of 100 produced a lowest fitness value of around 420,000, which is much smaller than that produced by the population of 10 at 640,000. This, combined with the fact that elitism is in play, means that the population of 100 has a much greater chance of producing good keyboards at an earlier stage than the population of 10.

Additionally, I am pretty sure that the evolution ran with a population of 10 encountered the problem of premature convergence, since after a few generations many of the keyboard layouts were similar and some were exactly the same. This is likely due to the fact that my selection function only takes into consideration the to 50% of keyboards in the population, meaning there were only 5 keyboards from it to choose from. This of course, is a problem since there is no diversity in the population for the keyboards to be optimised correctly and will likely result in the best keyboards only crossing over with themselves, which is not beneficial. In fact, if you look at the graph of lowest distances for each population, you will notice that there are more “slopes” downwards throughout the population of 100 than there are in the population of 10. This is a clear indication that the population of 100 optimised more than that of 10 did.

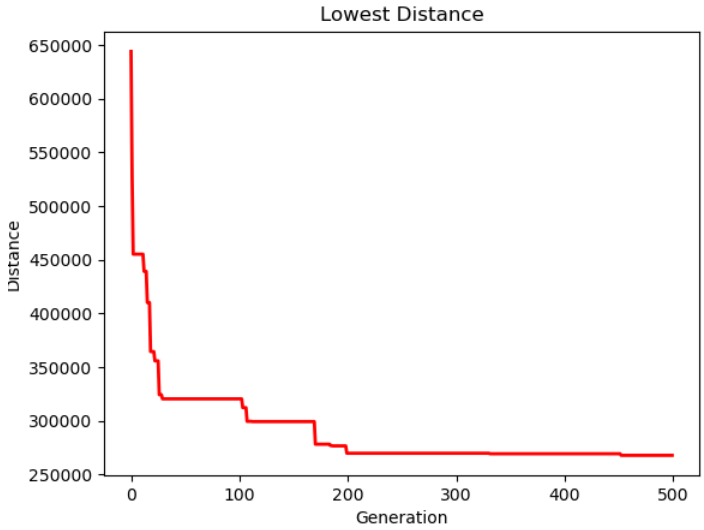
This is likely the biggest factor as to why the population of 10 did not produce a keyboard better than that of the population of 100, however I knew that the best way to solve this was to just increase the population size, which I did and it managed to produce the keyboard presented previously and I did not notice any premature convergence while it was running.

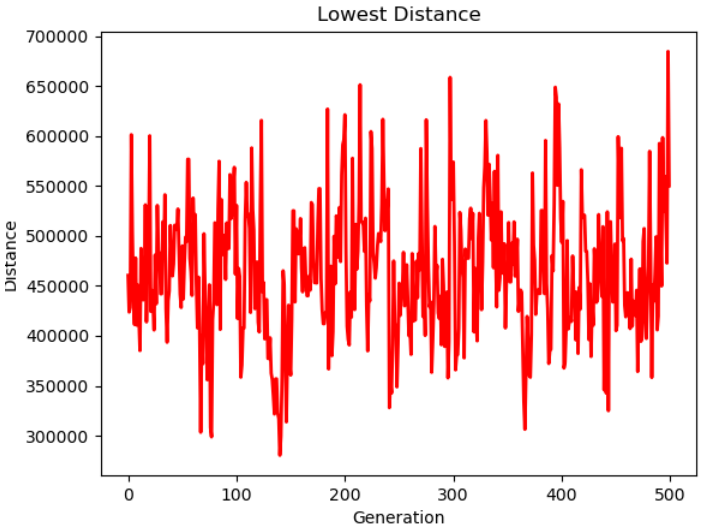
On another note, if we take a look at the graphs for average distance of both you can easily notice that the one for population of 10 is much more varied than the other. In fact, it managed to produce 2 outliers, one in a positive way (under 400,000 average distance) and another in a negative way (800,000 average distance). I think this is also due to the problem of premature convergence, since the keyboards are very similar to each other, meaning that if they are bad, when they crossover they will likely be worse and if they are good, when they crossover they will probably be better.

I never ran the evolution with a population size over 100 as it would have taken too long to finish, running it with a population of 100 for 500 generations took 10+ hours.

The Effect of Elitism

When observing the below graphs for lowest distance, the results are incredibly different when running the evolution with and without elitism. When running without elitism, the lowest fitness value is not immediately passed on to the next generation, hence resulting in the lowest distance varying tremendously. The lowest distance also ends up being significantly higher than when it was after the first generation, which leads me to believe that there is a problem involving premature convergence here. Either way, it is undoubtable that elitism helps the best chromosomes go through to the next generation and allow for the better chromosomes to create offsets of themselves.





Conclusion and Final Thoughts

In the context of the AI Bachelors course, I feel that this is the first real taste we have had in designing something that is actually closely related to artificial intelligence. It is something that I feel is quite belated, as I have been waiting to implement something like this ever since I started the course, however I am glad the opportunity finally came. Conducting this project opened my eyes as to why I've chosen this course as I found it very interesting learning in greater detail how AI algorithms such as genetic algorithms work and how powerful they can be if implemented correctly.

Based on the detailed examination and comparisons that I have done, I believe the genetic algorithm works well and provides well optimised outputs, however there is no doubt that it has some flaws for reasons which I have mentioned throughout this report, although overall I am happy with the results it has produced.

References

[1] https://www.gutenberg.org/cache/epub/5197/pg5197.txt

[2] http://www.ijmlc.org/papers/146-C00572-005.pdf