L-Università ta' Malta

Machine Learning: Introduction to Classification, Search and Optimisation

ICS2207

Programmed using Python

By Mark Dingli
20703H

Table of contents

Statement of Completion

Item	Completed (Yes/No/Partial)
Implement 'base' genetic algorithm	COMPLETED
Two-point crossover	PARTIAL
Implemented a mutation operation	COMPLETED
Elitism	COMPLETED
Good evaluation and discussion	COMPLETED
Attempted but Single Point Crossover was used	

[illegible]

Student's Signature

Introduction

Keyboards are currently the most used way of communicating with computers. Despite being the most widely used keyboard layout, QWERTY has had several concerns raised about its efficiency.
A genetic algorithm is a type of optimization algorithm that is inspired by natural evolution. It is used to solve optimization problems by iteratively improving a set of solutions (called a population) through a process of selection, crossover (recombination), and mutation. The algorithm begins with a population of randomly generated solutions and then applies these operators to create a new generation of solutions. This process is repeated until a satisfactory solution is found or the maximum number of iterations is reached.
In this case, the genetic algorithm is being used to find an optimal keyboard layout. The initial population consists of randomly generated keyboard layouts, and the fitness of each layout is determined by how well it satisfies certain criteria, such as the distance between keys that are pressed. The genetic operators are then applied to create a new generation of keyboard layouts, and the process is repeated until a satisfactory layout is found.
As a result, this Jupyter Notebook describes the process of developing a Genetic Algorithm with the goal of providing a more appropriate and cohesive layout proposal for the English (United Kingdom) keyboard layout, with an emphasis on ergonomics and user productivity. By using a genetic algorithm and a statistical examination of the letters and sequences of the most used words in both written and spoken English, a layout 45% better than the QWERTY keyboard was created.

Design

This section details the procedures used to create the algorithm. The selection of a keyboard layout that is appropriate for certain factors is predicated on the search for a specific arrangement among a collection of available key combinations. Given the extent of the solution space, a Genetic Algorithm stands out as a suitable approach for conducting this search.
This section describes all the sections of code found in the Jupyter Notebook.

Imports

```
import random
import math
from pprint import pprint
from PIL import Image, ImageDraw, ImageFont
import matplotlib.pyplot as plt
import json
```

Description:

This code imports several libraries that can be used for different purposes. The *random* library can be used to generate random numbers. The *math* library contains mathematical functions such as trigonometric functions and logarithms. The *pprint* library provides a way to "pretty print" data structures such as lists and dictionaries. The *PIL* library (short for "Python Imaging Library") provides tools for working with images. The *matplotlib* library is a popular plotting library for creating graphs and charts in Python. Finally, the *json* library allows you to work with data in the JSON format.

The Keyboard that will be optimized

Code:

```
# String for all the QWERTY keyboard characters
keyboard_characters = 'QAZWSXEDCRFVTGBYHNUJMIK,OL.P;/'

# Prints the number of characters
print("\n Total number of characters: ", len(keyboard_characters))

# Opens keyboard image
with Image.open("Images/keyboard.jpg").convert("RGBA") as start:
    start_characters = Image.new("RGBA", start.size, (255, 255, 255, 0))

    # Import font for characters
    font = ImageFont.truetype("Fonts/open_sans_bold.ttf", 35)
    # Keyboard number
    font_index = ImageFont.truetype("Fonts/open_sans_bold.ttf", 11)
    draw_image = ImageDraw.Draw(start_characters)
    # Offset for the 3 rows of characters (x-axis)
    row_offsets = [110, 135, 175]

    # for loop to get initial keyboard coordinates
    for i in range(30):
        keyboard_row = i%3
        keyboard_column = math.floor(i/3)
        x = keyboard_column*58 + row_offsets[keyboard_row]
        y = keyboard_row*65 + 85
        character_coords = (x, y)
        index_coords = (x-3, y-3)

        # Keyboard Characters
        draw_image.text(index_coords, str(i), font=font_index, fill=(0, 0, 0, 255))
        # Keyboard Numbers
         draw_image.text(character_coords, keyboard_characters[i], font=font, fill=(0,
0, 0, 255))

    # Displays the keyboard with the characters and respective number
    print_keyboard = Image.alpha_composite(start, start_characters)
    display(print_keyboard)
```

Description:

The given code generates an image of a keyboard with the QWERTY keyboard characters displayed on the keys. The code uses the 'keyboard_characters' string to store all the characters, and then uses a for loop to iterate over the characters and place them on the corresponding keys in the keyboard image. It also adds a number to each key to indicate its position in the 'keyboard_characters' string.

Output:

```
Total number of characters: 30
```

Figure 1 shows the English (United Kingdom) QWERT keyboard layout that was outputted. This is the keyboard that will optimised.

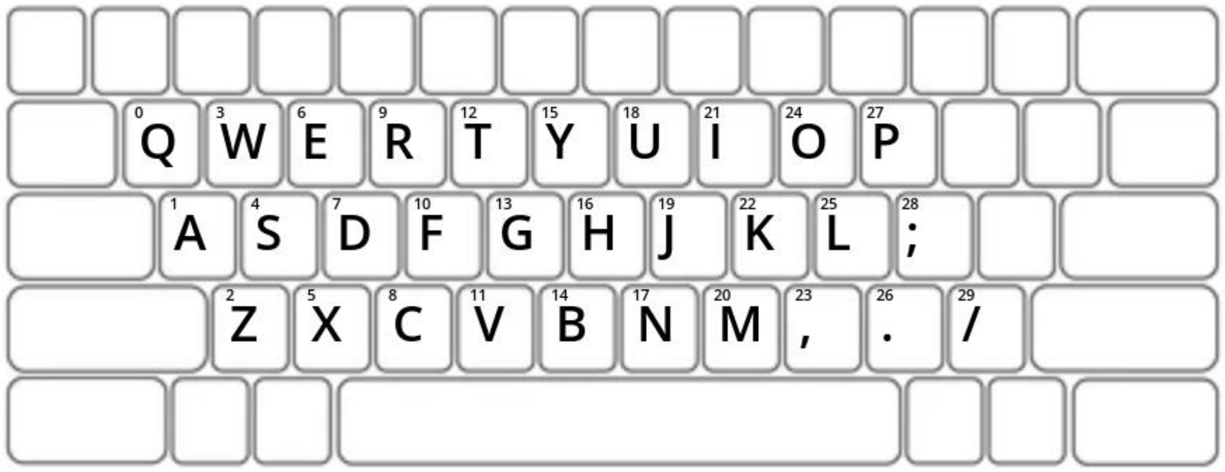

Figure 1

Getting initial keyboard coordinates

Code:

```
# Setting variables for keyboard offset
key_width = 94
middle_offset = 24
bottom_offset = 71
offsets = [0, middle_offset, bottom_offset]

# coordinates dictionary to represent QWERTY Keyboard
coordinates = {}

# for loop to get initial keyboard coordinates
for i in range(30):
    row = i%3
    column = math.floor(i/3)
    x = column*key_width+offsets[keyboard_row]
    y = row*key_width
    # list to store QWERTY keyboard coordinates
    coordinates[i] = (x,y)

# Prints coordinates
Coordinates
```

Description:

The code above sets some initial variables for the keyboard, including the key width and offsets for the different rows. It then uses a for loop to iterate over the keys and calculate their coordinates. These coordinates are then stored in a dictionary, with the keys of the dictionary corresponding to the position of the character in the keyboard_characters string. Finally, the coordinates of all the keys are printed.

Output:

```
{0: (71, 0),          20: (635, 188),
 1: (71, 94),         21: (729, 0),
 2: (71, 188),        22: (729, 94),
 3: (165, 0),         23: (729, 188),
 4: (165, 94),        24: (823, 0),
 5: (165, 188),       25: (823, 94),
 6: (259, 0),         26: (823, 188),
 7: (259, 94),        27: (917, 0),
 8: (259, 188),       28: (917, 94),
 9: (353, 0),         29: (917, 188)}
 10: (353, 94),
 11: (353, 188),
 12: (447, 0),
 13: (447, 94),
 14: (447, 188),
 15: (541, 0),
 16: (541, 94),
 17: (541, 188),
 18: (635, 0),
 19: (635, 94),
```

Getting First and last keys' Coordinates

Code:

```
# Function that gets distances between keys
def keys_distance(first, second):
    return math.hypot(second[0] - first[0], second[1] - first[1])

# Outputting first (Q) and last (/) keys
print("\nFirst key: " ,keys_distance(coordinates[0], coordinates[0]))
print("\nLast keys: " ,keys_distance(coordinates[0], coordinates[29]))
```

Description:

The code above defines a function called ‘keys_distance’ that calculates the distance between two keys on the keyboard. The function takes in two key coordinates and uses the **Pythagorean theorem** to calculate the distance between them. It then outputs the distance between the first (Q) and last (/ key) using the ‘keys_distance’ function and the coordinates of the keys from the ‘coordinates’ dictionary.

Output:

```
First key: 0.0

Last keys: 866.6371789855315
```

Keyboard for 8 Finger placement

Figure 2 shows the English (United Kingdom) QWERTY keyboard layout and the starting keys for an 8-finger layout.

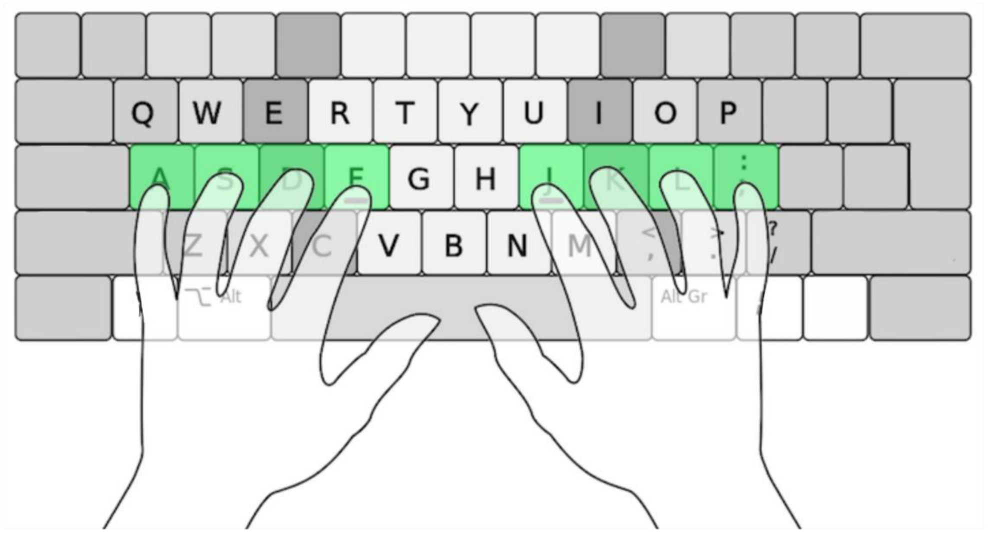

Figure 2

Fingers and every corresponding keys

Figure 3 shows the English (United Kingdom) QWERTY keyboard layout as well as the 8-finger hand position and the keys that correspond to each finger.

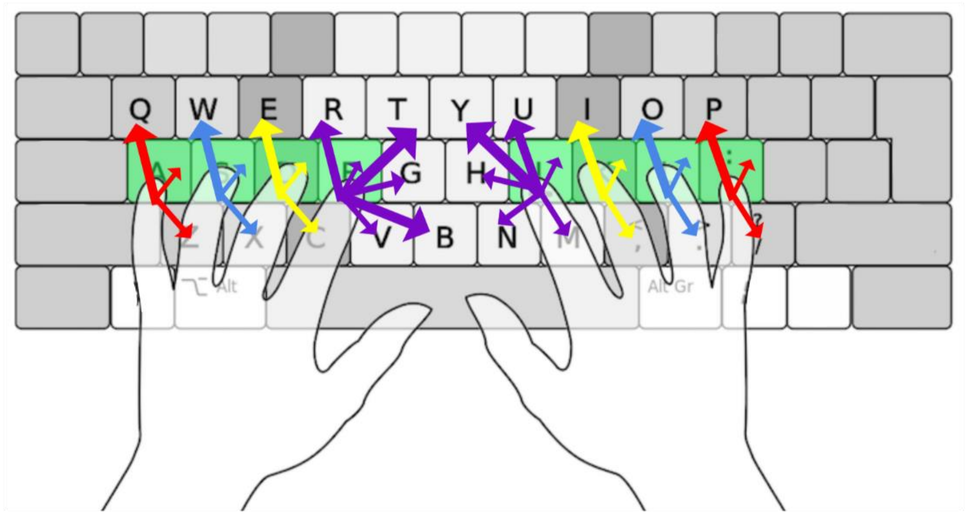

Figure 3

- Left Pinky Finger → Q - A - Z
- Left Ring Finger → W - S - X
- Left Middle Finger → E - D - C - R - F - V
- Left Pointer Finger → T - G - B - Y - H - N
- Right Pointer Finger → U - J - M
- Right Middle Finger → I - K - ,
- Right Ring Finger → O - L - .
- Right Pinky Finger → P - ; - /

Code:

```
# Placement for every finger and corresponding letters
# Corresponding keys for every finger
corresponding_keys = [[0,1,2], [3,4,5], [6,7,8], [9,10,11,12,13,14], [15,16,17,18,19,20],
[21,22,23], [24,25,26], [27,28,29]]

# Getting Distances for every finger and corresponding key
finger_distances = {i: {} for i in range(30)}

for total_keys in corresponding_keys:
    for i in total_keys:
        for j in total_keys:
            finger_distances[i][j] = keys_distance(coordinates[i], coordinates[j]) / key_width

# Output all distances
finger_distances
```

Description:

The code above sets up a list called ‘corresponding_keys’ that defines which keys correspond to each finger. It then creates a dictionary called ‘finger_distances’ that will store the distances between each key that corresponds to a finger. A nested for loop is used to iterate over the keys and calculate the distances between them using the keys_distance function. Finally, the ‘finger_distances’ dictionary is printed. This dictionary can be used to determine how far a finger needs to move to reach different keys on the keyboard.

Output:

```
{0: {0: 0.0, 1: 1.0, 2: 2.0},
1: {0: 1.0, 1: 0.0, 2: 1.0},
2: {0: 2.0, 1: 1.0, 2: 0.0},
3: {3: 0.0, 4: 1.0, 5: 2.0},
4: {3: 1.0, 4: 0.0, 5: 1.0},
5: {3: 2.0, 4: 1.0, 5: 0.0},
6: {6: 0.0, 7: 1.0, 8: 2.0},
7: {6: 1.0, 7: 0.0, 8: 1.0},
8: {6: 2.0, 7: 1.0, 8: 0.0},
9: {9: 0.0,
10: 1.0,
11: 2.0,
12: 1.0,
13: 1.4142135623730954,
14: 2.23606797749979},
10: {9: 1.0,
10: 0.0,
11: 1.0,
12: 1.4142135623730954,
13: 1.0,
14: 1.4142135623730954},
11: {9: 2.0,
10: 1.0,
11: 0.0,
12: 2.23606797749979,
13: 1.4142135623730954,
14: 1.0},
12: {9: 1.0,
10: 1.4142135623730954,
11: 2.23606797749979,
12: 0.0,
13: 1.0,
14: 2.0},
13: {9: 1.4142135623730954,
10: 1.0,
11: 1.4142135623730954,
12: 1.0,
13: 0.0,
14: 1.0},
14: {9: 2.23606797749979,
10: 1.4142135623730954,
11: 1.0,
12: 2.0,
13: 1.0,
14: 0.0},
15: {15: 0.0,
16: 1.0,
17: 2.0,
18: 1.0,
19: 1.4142135623730954,
20: 2.23606797749979},
16: {15: 1.0,
16: 0.0,
17: 1.0,
18: 1.4142135623730954,
19: 1.0,
20: 1.4142135623730954},
17: {15: 2.0,
16: 1.0,
17: 0.0,
18: 2.23606797749979,
19: 1.4142135623730954,
20: 1.0},
18: {15: 1.0,
16: 1.4142135623730954,
17: 2.23606797749979,
18: 0.0,
19: 1.0,
20: 2.0},
19: {15: 1.4142135623730954,
16: 1.0,
17: 1.4142135623730954,
18: 1.0,
19: 0.0,
20: 1.0},
20: {15: 2.23606797749979,
16: 1.4142135623730954,
17: 1.0,
18: 2.0,
19: 1.0,
20: 0.0},
21: {21: 0.0, 22: 1.0, 23: 2.0},
22: {21: 1.0, 22: 0.0, 23: 1.0},
23: {21: 2.0, 22: 1.0, 23: 0.0},
24: {24: 0.0, 25: 1.0, 26: 2.0},
25: {24: 1.0, 25: 0.0, 26: 1.0},
26: {24: 2.0, 25: 1.0, 26: 0.0},
27: {27: 0.0, 28: 1.0, 29: 2.0},
28: {27: 1.0, 28: 0.0, 29: 1.0},
29: {27: 2.0, 28: 1.0, 29: 0.0}}
```

Setting initial positions of fingers

Code:

```
# Initial keys postion of fingers when using 8 fingers
initial_key_pos = [1, 4, 7, 10, 19, 22, 25, 28]

initial_keys = {}
for i, keys in enumerate(corresponding_keys):
    for key in keys:
        initial_keys[key] = initial_key_pos[i]

# Outputs the initial keys the fingers will be on and the top and bottom key that corresponds to
the finger
initial_keys
```

Description:

Outputs the initial keys the fingers will be on and the top and bottom key that corresponds to the finger:
Example: Left Pinky finger will be on A(1), top key will be Q(0) and bottom key is Z(2). Therefore 1 corresponds to 0 and 2.
This code defines a list called 'initial_key_pos' that represents the initial position of each finger on a QWERTY keyboard. It then creates a dictionary called 'initial_keys', which maps each key on the keyboard to the finger that is initially positioned on that key.

Output:

```
{0: 1,
1: 1,
2: 1,
3: 4,
4: 4,
5: 4,
6: 7,
7: 7,
8: 7,
9: 10,
10: 10,
11: 10,
12: 10,
13: 10,
14: 10,
15: 19,
16: 19,
17: 19,
18: 19,
19: 19,
20: 19,
21: 22,
22: 22,
23: 22,
24: 25,
25: 25,
26: 25,
27: 28,
28: 28,
29: 28}
```

Returning a gene as a keyboard Dictionary

Including letters when Shift is pressed

Code:

```
# Define the default keyboard characters
keyboard_characters = 'QAZWSXEDCRFVTGBYHNUJMIK,OL.P;/'

def genome_keyboard(gene):
    # Create an empty dictionary for mapping the keyboard characters to their indices
    keyboard_dict = {}

    # Loop through the characters in the gene and add them to the dictionary
    for i, char in enumerate(gene):
        keyboard_dict[char] = i

    # If the ',' character is in the gene, add it to the dictionary with the special '<' key
    if ',' in gene:
        keyboard_dict['<'] = gene.index(',')
    # If the '.' character is in the gene, add it to the dictionary with the special '>' key
    if '.' in gene:
        keyboard_dict['>'] = gene.index('.')
    # If the ';' character is in the gene, add it to the dictionary with the special ':' key
    if ';' in gene:
        keyboard_dict[':'] = gene.index(';')
    # If the '/' character is in the gene, add it to the dictionary with the special '?' key
    if '/' in gene:
        keyboard_dict['?'] = gene.index('/')

    # Return the dictionary of keyboard characters and their indices
    return keyboard_dict

# Output the dictionary of keyboard characters and their indices when the shift key is pressed
genome_keyboard(keyboard_characters)
```

Description:

This code defines a function called genome_keyboard that takes a string gene as input and returns a dictionary that maps each character in the gene to its index in the gene. The function first creates an empty dictionary called ‘keyboard_dict’, and then loops through the characters in the gene and adds them to the dictionary. If the gene contains any of the special characters ',', '.', ';', or '/', the function adds these characters to the dictionary using the special keys '<', '>', ':', and '?' respectively. The function then returns the ‘keyboard_dict’ dictionary.

An example of how this function can be used is shown in the last line of the code, where the genome_keyboard function is called with the default keyboard characters as the argument. This outputs a dictionary that maps each of the default keyboard characters to its index in the keyboard_characters string.

Output:

```
{'Q': 0,
'A': 1,
'Z': 2,
'W': 3,
'S': 4,
'X': 5,
'E': 6,
'D': 7,
'C': 8,
'R': 9,
'F': 10,
'V': 11,
'T': 12,
'G': 13,
'B': 14,
'Y': 15,
'H': 16,
'N': 17,
'U': 18,
'J': 19,
'M': 20,
'I': 21,
'K': 22,
',': 23,
'<': 23,
'O': 24,
'L': 25,
'.': 26,
'>': 26,
'P': 27,
';': 28,
':': 28,
'/': 29,
'?': 29}
```

Importing the Dataset

Code:

```
# Converting all chracters to upper case
#with open('Dataset/text.txt', 'r') as data_file:

    # open output.txt file in append mode
    #with open('Dataset/text.txt', 'a') as output_file:

        # read each line from data.txt
        #for line in data_file:

            # change case for the line and write
            # it into output file
            #output_file.write(line.upper())

#---------------------------------------------------------------------------------
#Removing line breaks

# first get all lines from file
#with open('Dataset/text.txt', 'r') as f:
#   lines = f.readlines()

# remove spaces
#lines = f.replace('\n', '')

# finally, write lines in the file
#with open('Dataset/text.txt', 'w') as f:
#   f.writelines(lines)

#---------------------------------------------------------------------------------
final_text = ''

count = 0

#open data file in read mode
with open("Dataset/english.txt", "r") as file:

    #read the content of file
    dataset = file.readlines()

    #iterate through every letter and add to full_text and increment counter
    for d in dataset:
            for words in d:
                    final_text += ' ' + words
                    count += 1

#print the counter
print('Number of characters: ',count)
#print the length(number of items in a list) of the data
print('Length of text: ',len(final_text))

#close the file
file.close()
```

Description:

For this assignment, The **Longman Communication 3000** data set was used. This dataset has a list of 3000 words that are the most frequent words in both spoken and written English. This is based on statistical analysis of the 390 million words contained in the Longman Corpus Network.

Reference: https://www.lextutor.ca/freq/lists_download/longman_3000_list.pdf

Analysis of the Longman Corpus Network shows that these 3000 most frequent words in spoken and written English account for 86% of the language. This means that by knowing this list of words, a learner of English is in a position to understand 86% or more of what they read. It is important to note that all the extra paragraphs, letters, numbers and images were removed from the dataset.
The code converts all the characters in the 'text.txt' file to uppercase and then removes line breaks. It does this by first opening the text.txt' file in read mode and then appending the uppercase version of each line to the same file. It then reads the entire file into a list of lines, removes the line breaks, and writes the modified lines back to the file. Finally, the code reads the file again and concatenates all the characters into a single string called 'final_text'. It also counts the total number of characters and prints this count along with the length of the 'final_text' string.

Output:

```
Number of characters: 35890
Length of text: 71780
```

Calculating the total keyboard distance travelled

Code:

```
# Function that calculates the total keyboard distance travelled
def total_distance_travelled(inputted_string, keyboard_dict):
    # Replace spaces in the inputted string with no space
    inputted_string = inputted_string.replace(' ', '')

    # Get the initial position of the first character
    first_character = inputted_string[0]
    initial_pos = keyboard_dict[first_character]

    # Check if the initial position has a home key
    initial_home_key = initial_keys.get(initial_pos, None)
    if initial_home_key is not None:
        # If it has a home key, add the distance from the home key to the initial
position
        total_distance = finger_distances[initial_home_key][initial_pos]
    else:
        # If it doesn't have a home key, add 0 distance
        total_distance = 0

    # Loop through the characters in the inputted string
    for i in range(0, len(inputted_string)-1):
        current_character = inputted_string[i]
        next_character = inputted_string[i+1]

        # Get the current and next position of the characters
        current_position = keyboard_dict[current_character]
        next_position = keyboard_dict[next_character]

        # Check if the distance between the current and next position is in the
finger_distances dictionary
        if current_position in finger_distances and next_position in
finger_distances[current_position]:
            # If it is, add the distance to the total distance
            total_distance += finger_distances[current_position][next_position]
        else:
            # If not, check if the next position has a home key
            home_key = initial_keys.get(next_position, None)
            if home_key is not None:
                # If it has a home key, add the distance from the home key to the next
position
                total_distance += finger_distances[home_key][next_position]
            else:
                # If it doesn't have a home key, add 0 distance
                total_distance += 0

    # Return the total distance
    return total_distance

# Test the function
total_distance_travelled('THIS IS A TEST FOR DISTANCE.',
genome_keyboard(keyboard_characters))
```

Description:

Important to note that the spaces, although characters do not affect the total distance travelled on the keyboard. The 'total_distance_travelled' function calculates the total distance that a finger needs to travel to type a given string on a keyboard with a given layout (specified by a dictionary generated by the 'genome_keyboard' function). The function first removes any spaces from the input string and then calculates the initial distance that the finger needs to travel to reach the first character in the string. It then uses a for loop to iterate over the characters in the string and calculate the distance between each consecutive pair of characters. If the distance between two characters is not specified in the 'finger_distances' dictionary, the function calculates the distance by first finding the home key for the next character and then using the distance between the home key and the next character. The output tests the function.

Output:

The function returns the total distance that the finger needs to travel to type the entire string:

```
19.485281374238575
```

Total Keyboard distance travelled for the inputted dataset
Code:

```
total_distance_travelled(final_text, genome_keyboard(keyboard_characters))
```

Description:

The given code calculates the total distance that a finger would need to travel to type the entire 'final_text' string on a QWERTY keyboard using the 'total_distance_travelled' function.

Output:

```
19200.22075411139
```

Genetic Algorithm Implementation

Code:

```
# Creating first generation chromosome
def initial_generation(amount):
    #create list for keyboard characters
    keyboard_characters = list('QAZWSXEDCRFVTGBYHNUJMIK,<OL.>P;:/?')
    #creating a list to hold new population
    chromosome = []

    #creating a random Chromosome by shuffeling the keyboard population
    for i in range(amount):
        #slice every element in the list
        random_gene = keyboard_characters[:]

        #randomise keyboard characters
        random.shuffle(random_gene)
        #append random character to list
        chromosome.append(random_gene)
    return chromosome

#testing function
test_chromosomes = initial_generation(2)
#print
test_chromosomes
```

Description:

The ‘initial_generation’ function creates a list of random keyboard layouts, also known as chromosomes. The function takes in a single parameter, ‘amount’, which specifies the number of chromosomes to generate. It first creates a list of the QWERTY keyboard characters, then uses a for loop to generate a number of chromosomes (in this case 2) by randomly shuffling the list of keyboard characters. The function returns the list of randomly generated chromosomes.

Output:

```
[['H',
',',
'A',
'I',
'P',
'W',
'R',
'M',
'L',
'Q',
'F',
'.',
'C',
'G',
'U',
'E',
'B',
'T',
'V',
'O',
';',
'K',
'X',
'S',
'/',
'N',
'J',
'Y',
'Z',
'D'],
['W',
'D',
'L',
'S',
'M',
'F',
'U',
'N',
'B',
'A',
'P',
'.',
'H',
'E',
'G',
'C',
'O',
',',
'/',
'X',
'V',
';',
'Z',
'R',
'T',
'K',
'J',
'I',
'Y',
'Q']]
```

Combining two keyboards

- **Crossover Function**
- **Mutation Function**

Code:

```
# Crossover function that creates a child keyboard from two parent keyboards
def crossover_function(keyboard_1, keyboard_2, mutation_rate):
    # Get the size of the keyboard
    keyboard_size = len(keyboard_1)

    # Generate a random starting index and length for the child keyboard
    identiy_x = random.randint(0, keyboard_size-1)
    keyboard_length = random.randint(0, keyboard_size-1)

    # Create an empty child keyboard
    child = ['_' for i in range(keyboard_size)]

    # Fill the child keyboard with characters from the first parent keyboard
    for i in range(keyboard_length):
        # If the index is greater than the size of the keyboard, set it to 0
        if identiy_x > keyboard_size-1:
            identiy_x = 0
        # Set the character at the current index in the child to the character at the 
same index in the first parent
        child[identiy_x] = keyboard_1[identiy_x]
        # Increment the index
        identiy_x += 1

    # Set the index for filling the child keyboard with characters from the second 
parent
    child_identity_x = identiy_x

    # While the child keyboard has empty spaces, fill them with characters from the 
second parent
    while '_' in child:
        # If the index is greater than the size of the keyboard, set it to 0
        if identiy_x > keyboard_size-1:
            identiy_x = 0
        if child_identity_x > keyboard_size-1:
            child_identity_x = 0
        # Get the character at the current index in the second parent
        char = keyboard_2[identiy_x]
        # If the character is already in the child, skip it
        if char in child:
            identiy_x += 1
            continue
        # Set the character at the current index in the child to the character from 
the second parent
        child[child_identity_x] = keyboard_2[identiy_x]
        # Increment the index for filling the child
        child_identity_x += 1
        # Increment the index for getting characters from the second parent
        identiy_x += 1

    # Generate a random probability
    probability = random.random()

    # If the probability is less than the mutation rate, mutate the child
    if probability < mutation_rate:
        # Generate two random indices in the child keyboard
        pointer_1 = random.randint(0, 29)
        pointer_2 = random.randint(0, 29)
        # Get the characters at the two indices
        gene_pair_1 = child[pointer_1]
        gene_pair_2 = child[pointer_2]
        # Swap the characters at the two indices
        child[pointer_1] = gene_pair_2
        child[pointer_2] = gene_pair_1

    # Return the child keyboard
    return child

# Output when mating two chromosomes (keyboards) together, includes mutation rate
crossover_function(test_chromosomes[0], test_chromosomes[1], 0.1)
```

Description:

This code defines a function called 'crossover_function' that takes in three arguments: 'keyboard_1, keyboard_2', and 'mutation_rate'. The function first determines the size of the keyboards by getting the length of 'keyboard_1'. It then generates a random starting index and length for the child keyboard, which is initially empty. Next, the function fills the child keyboard with characters from 'keyboard_1' by iterating over a range of indices determined by the previously generated length and incrementing the index each iteration. The function then repeats this process with 'keyboard_2', filling in any remaining empty spaces in the child keyboard with characters from 'keyboard_2'. Finally, the function has a probability of mutating the child keyboard based on the 'mutation_rate' argument. If the child is mutated, two random indices are selected and the characters at those indices are swapped. The child keyboard is then returned. A mutation rate of 0.1 was used since successful mutation rates are typically reported from 0.5% to 1.0%. The function returns the resulting child string.

Output:

```
['A',
 'P',
 'H',
 ',',
 '/',
 'X',
 'R',
 'M',
 'L',
 'Q',
 'F',
 '.',
 'C',
 'G',
 'U',
 'E',
 'B',
 'T',
 'V',
 'O',
 ';',
 'Z',
 'K',
 'J',
 'I',
 'Y',
 'W',
 'D',
 'S',
 'N']
```

<u>Fitness Function (Elitism)</u>
<u>Code:</u>

```
# This function takes in a list of chromosomes and returns a dictionary of their
fitness values,
# as well as a sorted list of the indices of the chromosomes in ascending order of
their fitness
def elitism(chromosome):
    # Initialize an empty dictionary to store the fitness values of the chromosomes
    evaluation = {}

    # Iterate over the chromosomes, calculating their fitness values and storing them
in the dictionary
    for i, genome in enumerate(chromosome):
        # Calculate the fitness value of the current chromosome by getting the
keyboard it produces
        # and then measuring the total distance travelled to type the final text using
this keyboard
        fittest_keyboard = genome_keyboard(genome)
        distance = total_distance_travelled(final_text, fittest_keyboard)
        evaluation[i] = distance

    # Sort the chromosomes by their fitness values and return a list of their indices
in this order
    sorted_evaluation = [i for i, j in sorted(evaluation.items(), key=lambda item:
item[1])]

    # Return the dictionary of fitness values and the sorted list of indices
    return evaluation, sorted_evaluation

# Call the elitism() function on the test chromosomes
elitism(test_chromosomes)
```

Description:

This code defines a function called ‘elitism’ that takes in a list of chromosomes called ‘chromosome’ and returns a dictionary of their fitness values and a sorted list of the indices of the chromosomes in ascending order of their fitness. The function first initializes an empty dictionary called ‘evaluation’ to store the fitness values of the chromosomes. It then iterates over the chromosomes, calculating their fitness values using the ‘genome_keyboard’ and ‘total_distance_travelled’ functions and storing them in the ‘evaluation’ dictionary. The function then sorts the chromosomes by their fitness values and returns a list of their indices in this order. Finally, the function returns both the ‘evaluation’ dictionary and the sorted list of indices.

Output:

```
({0: 20475.39687397908, 1: 19076.135277408663}, [1, 0])
```

Creating the next Generation

Code:

```
# This function takes in a list of chromosomes, a sorted list of their indices in
ascending order of fitness,
# the size of the population, and the mutation rate, and produces a new generation of
chromosomes by applying
# crossover and mutation
def next_generation(chromosome, sorted_evaluation, p_size, mutation_rate):
    # Initialize an empty list to store the new generation of chromosomes
    new_generation = []

    # Sort the chromosomes by their fitness values
    sorted_population = []
    for i in sorted_evaluation:
        sorted_population.append(chromosome[i])

    # Add the 10% most fit chromosomes to the new generation without modification
    for i in range(int(p_size*0.1)):
        new_generation.append(sorted_population[i])

    # Generate the remaining 90% of the new generation by applying crossover and
mutation
    for _ in range(int(p_size*0.9)):
        # Choose two parent chromosomes from the top 50% of the sorted population
        population1 = random.choice(sorted_population[:int(p_size*0.5)])
        population2 = random.choice(sorted_population[:int(p_size*0.5)])

        # Generate a child by combining the characters of the two parent chromosomes
        child = crossover_function(population1, population2, mutation_rate)

        # Add the child to the new generation
        new_generation.append(child)

    # Return the new generation of chromosomes
    return new_generation

# Generate a new generation of chromosomes using the test chromosomes, the sorted
evaluation of the test chromosomes,
# a population size of 10, and a mutation rate of 0.1
new_generations_output = next_generation(test_chromosomes, sorted_evaluation=[1, 0],
p_size=10, mutation_rate=0.1)

# Print the new generation of chromosomes
print(new_generations_output)
```

Description:

This code defines a function called ‘next_generation’ that takes in a list of chromosomes called chromosome, a sorted list of their indices called ‘sorted_evaluation’, the size of the population called ‘p_size’, and the mutation rate called mutation_rate. The function produces a new generation of chromosomes by applying crossover and mutation. The function first initializes an empty list called ‘new_generation’ to store the new generation of chromosomes.

It then sorts the chromosomes by their fitness values by creating a new list called ‘sorted_population’ and appending the chromosomes in the order of their indices in ‘sorted_evaluation’. The function then adds the 10% most fit chromosomes to ‘new_generation’ without modification. Next, the function generates the remaining 90% of the new generation by applying crossover and mutation. It does this by choosing two parent chromosomes from the top 50% of the sorted population and generating a child by combining the characters of the two parent chromosomes using the ‘crossover_function’. The function then adds the child to ‘new_generation’. Finally, the function returns ‘new_generation’.

<u>Output:</u>

```
[['W', 'D', 'L', 'S', 'M', 'F', 'U', 'N', 'B', 'A', 'P', '.', 'H', 'E', 'G', 'C', 'O',
',', '/', 'X', 'V', ';', 'Z', 'R', 'T', 'K', 'J', 'I', 'Y', 'Q'], ['W', 'D', 'L', 'P',
'R', 'M', 'F', '.', 'C', 'G', 'U', 'E', 'B', 'T', 'V', 'O', ';', 'X', 'S', '/', 'N',
'Z', 'H', ',', 'A', 'K', 'J', 'I', 'Y', 'Q'], ['W', 'M', 'L', 'Q', 'F', 'U', 'S', 'N',
'B', 'A', 'P', '.', 'H', 'E', 'G', 'C', 'O', ',', 'V', 'X', '/', ';', 'Z', 'R', 'T',
'K', 'J', 'Y', 'D', 'I'], ['Q', 'D', 'L', 'S', 'M', 'F', 'U', 'N', 'B', 'A', 'P', '.',
'H', 'E', 'G', 'C', 'O', 'T', 'V', ';', 'K', 'X', '/', 'J', 'Y', 'Z', ',', 'I', 'W',
'R'], ['W', 'D', 'L', 'S', 'M', 'F', 'U', 'N', 'B', 'A', 'P', '.', 'H', 'E', 'G', 'C',
'O', ',', '/', 'X', 'V', ';', 'Z', 'R', 'T', 'K', 'J', 'I', 'Y', 'Q'], ['W', 'D', 'L',
'S', 'M', 'F', 'U', 'N', 'B', 'A', 'P', '.', 'H', 'E', 'G', 'C', 'O', ',', '/', 'X',
'V', ';', 'Z', 'R', 'T', 'K', 'J', 'I', 'Y', 'Q'], ['H', ',', 'Z', 'I', 'P', 'W', 'R',
'M', 'L', 'Q', 'F', '.', 'C', 'G', 'U', 'E', 'B', 'T', 'V', 'O', ';', 'K', 'X', 'S',
'/', 'N', 'J', 'Y', 'D', 'A'], ['H', ',', 'A', 'I', 'P', 'W', 'R', 'M', 'L', 'Q', 'F',
'.', 'C', 'G', 'U', 'E', 'B', 'T', 'V', 'O', ';', 'K', 'X', 'S', '/', 'N', 'J', 'Y',
'Z', 'D'], ['I', 'P', 'W', 'M', 'L', 'Q', 'F', '.', 'C', 'G', 'U', 'E', 'B', 'T', 'K',
'S', 'O', ',', '/', 'X', 'V', ';', 'Z', 'R', 'N', 'J', 'Y', 'D', 'H', 'A'], ['W', 'D',
'L', 'S', 'M', 'F', 'U', 'N', 'B', 'A', 'P', '.', 'H', 'E', 'G', 'C', 'V', 'O', ';',
'X', '/', 'Z', ',', 'R', 'T', 'K', 'J', 'I', 'Y', 'Q']]
```

Learning Code:

```
# The population size and the number of generations to run the genetic algorithm ()
POPULATION_SIZE = 300
GENERATIONS = 300

# The mutation rate to use
MUTATION_RATE = 0.1

# Initialize a dictionary to store information about each generation of the genetic
algorithm
learning = {
    'Generations': {}
}

# Generate the initial population of chromosomes
chromosome = initial_generation(POPULATION_SIZE)

# Run the genetic algorithm for the specified number of generations
for i in range(GENERATIONS):
    # Calculate the fitness values of the chromosomes and sort them by fitness
    evaluation, sorted_evaluation = elitism(chromosome)

    # Calculate the average fitness of the chromosomes in this generation
    sum_evals = 0
    for key in evaluation:
        sum_evals += evaluation[key]
    avgerage_evaluations = sum_evals/POPULATION_SIZE

    # Store information about this generation in the learning dictionary
    learning['Generations'][i] = {
        'Chromosome': chromosome,
        'Best': chromosome[sorted_evaluation[len(sorted_evaluation)-1]],
        'Min': evaluation[sorted_evaluation[0]],
        'Average': avgerage_evaluations
    }

    # Print information about this generation
    print('GEN: {}, AVG: {}, MIN: {}, BEST: {}'.format(i+1, avgerage_evaluations,
evaluation[sorted_evaluation[i]], chromosome[sorted_evaluation[i]]))

    # Generate the next generation of chromosomes using the current generation
    chromosome = next_generation(chromosome, sorted_evaluation, POPULATION_SIZE,
MUTATION_RATE)

# Save the learning dictionary to a JSON file
with open('optimistaion.json', 'w') as fp:
    json.dump(learning, fp)
```

Description:

The code starts by generating an initial population of 300 chromosomes, which represent different keyboard layouts. It then runs for 300 generations, calculating the fitness values of the chromosomes in each generation and sorting them by their fitness values. For each generation, the code calculates the average fitness of the chromosomes, stores information about the generation in a learning dictionary (including the chromosomes, the best chromosome, the minimum fitness value, and the average fitness value), and prints the values about the generation. It then generates the next generation of chromosomes using the current generation and the ‘next_generation()’ function, which applies crossover and mutation to produce new chromosomes. Finally, the code saves the learning dictionary to a JSON file.

Output:

```
GEN: 1, AVG: 18495.40993531623, MIN: 13181.092882558196, BEST: ['V', 'Y', 'K', 'F',
'T', 'Q', 'D', ',', 'I', 'M', 'A', 'H', 'U', '/', 'X', 'S', 'Z', 'G', 'W', 'N', 'R',
'B', 'E', 'C', '.', 'O', ';', 'J', 'L', 'P']
GEN: 2, AVG: 17118.653767277578, MIN: 12867.515480104994, BEST: ['D', 'O', 'Z', 'C',
'A', 'S', 'J', 'E', 'N', 'U', 'H', 'B', 'M', 'X', ',', '/', 'L', 'K', 'T', ';', 'V',
'W', 'R', 'Q', 'P', 'I', 'G', 'F', '.', 'Y']
GEN: 3, AVG: 16411.20780164997, MIN: 12114.170529584178, BEST: ['/', 'P', 'W', 'K',
'A', ';', 'V', 'R', '.', 'S', 'T', 'M', 'D', 'C', ',', 'G', 'I', 'B', 'O', 'E', 'Z',
'X', 'Y', 'U', 'L', 'N', 'Q', 'J', 'H', 'F']
GEN: 4, AVG: 15913.10573189077, MIN: 12114.170529584178, BEST: ['/', 'P', 'W', 'K',
'A', ';', 'V', 'R', '.', 'S', 'T', 'M', 'D', 'C', ',', 'G', 'I', 'B', 'O', 'E', 'Z',
'X', 'Y', 'U', 'L', 'N', 'Q', 'J', 'H', 'F']
GEN: 5, AVG: 15429.318232757474, MIN: 11650.779736637205, BEST: ['W', 'O', 'F', 'L',
'A', ',', 'Y', 'T', 'Z', 'J', 'B', 'V', 'K', 'U', 'H', 'G', 'M', 'X', 'C', '/', 'S',
';', 'E', 'I', 'D', 'R', 'Q', '.', 'N', 'P']
GEN: 6, AVG: 15461.786526291784, MIN: 11650.779736637205, BEST: ['W', 'O', 'F', 'L',
'A', ',', 'Y', 'T', 'Z', 'J', 'B', 'V', 'K', 'U', 'H', 'G', 'M', 'X', 'C', '/', 'S',
';', 'E', 'I', 'D', 'R', 'Q', '.', 'N', 'P']
GEN: 7, AVG: 15175.293903447864, MIN: 11546.013975754258, BEST: ['M', 'P', 'S', ',',
'O', 'K', 'W', 'I', 'Z', '.', 'N', 'J', 'Y', 'T', 'X', 'D', 'A', 'F', 'C', 'U', 'H',
'Q', 'L', 'G', ';', 'E', 'V', '/', 'R', 'B']
GEN: 8, AVG: 14770.483651120849, MIN: 11400.448016211656, BEST: ['P', 'N', 'L', 'I',
'C', 'J', 'Y', 'E', '/', ',', 'O', 'T', ';', 'H', 'Q', 'G', 'W', 'F', 'D', 'A', '.',
'V', 'K', 'U', 'X', 'S', 'B', 'Z', 'R', 'M']
GEN: 9, AVG: 14849.487892866864, MIN: 11021.806416125364, BEST: ['W', 'O', 'F', 'L',
'A', ',', 'Y', 'T', 'K', 'V', 'U', 'B', 'H', 'G', 'J', 'Z', 'M', 'X', 'C', '/', 'S',
';', 'E', 'I', 'D', 'R', 'Q', '.', 'N', 'P']
.
.
.
GEN: 300, AVG: 8341.738908419009, MIN: 18226.10155011984, BEST: ['S', 'V', 'J', 'A',
'G', 'H', 'R', 'L', 'W', 'M', 'T', '.', 'D', ';', '/', 'Y', ',', 'B', 'I', 'Z', 'K',
'O', 'U', 'Q', 'E', 'X', 'F', 'N', 'P', 'C']
```

Visualising Results

Code:

```
# Import the json module for reading the data file
# and the matplotlib module for plotting the results
import json
import matplotlib.pyplot as plt

# Open the data file and read the learning data
with open('optimistaion.json') as fp:
    learning = json.load(fp)

# Initialize some variables for storing the results
last_distance = 1000000000
min_dists = []
avg_dists = []
generations = len(learning['Generations'])

# Loop through each generation and extract the minimum and average distances
for i in range(0, generations):
    # Get the minimum and average distances for this generation
    min_dist = learning['Generations'][str(i)]['Min']
    avg_dist = learning['Generations'][str(i)]['Average']

    # Add the distances to the list of results
    min_dists.append(min_dist)
    avg_dists.append(avg_dist)

# Create the plot with two subplots (one for the minimum distance and one for the
average distance)
fig, axes = plt.subplots(1, 2, figsize=(20, 6))

# Plot the minimum distances
axes[0].plot(min_dists, label='Lowest Distance')
axes[0].set_xlabel('Generations')
axes[0].set_ylabel('Distance')
axes[0].set_title('Best Distance')

# Plot the average distances
axes[1].plot(avg_dists, label='Average Distance', color='orange')
axes[1].set_xlabel('Generations')
axes[1].set_ylabel('Distance')
axes[1].set_title('Average Distance')

# Show the plot
plt.show()
```

Description:

This code reads the JSON file that contains the results of a genetic algorithm for optimizing a keyboard layout for typing a given text. It then extracts the minimum and average distances for each generation and stores them in separate lists. It then creates a plot with two subplots, one for the minimum distances and one for the average distances, and plots the results. Finally, it displays the plot.

Output:

Figure 4 shows the progress of the genetic algorithm over time. The x-axis represents the generation number, and the y-axis represents the distance (the lower the distance, the better the keyboard layout). The left subplot shows the minimum distance (the best distance) for each generation, while the right subplot shows the average distance for each generation. After a certain point, it is evident that the algorithm does not make any additional improvements. The plot allows you to see how the genetic algorithm improves the keyboard layout over time and how the average distance changes (reduces) as the algorithm progresses.

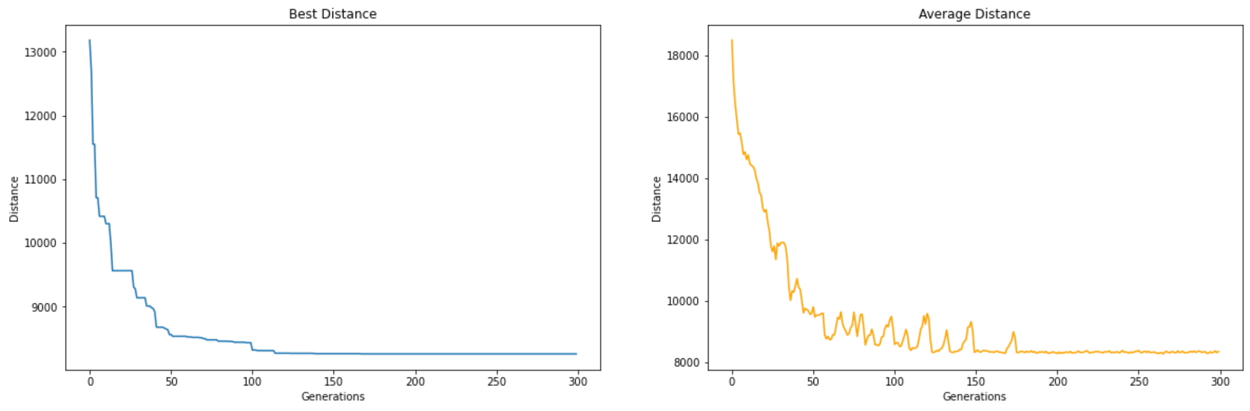

Figure 4

The final proposed optimised keyboard

Code:

```
# Get the keyboard characters from the last generation of the learning data
keyboard_characters = learning['Generations'][str(GENERATIONS-1)]['Best']

# Open the keyboard image and convert it to RGBA format
with Image.open("Images/keyboard.jpg").convert("RGBA") as base:
    # Create a new image with a transparent background
    txt = Image.new("RGBA", base.size, (255, 255, 255, 0))

    # Load the font for the keyboard characters and the font for the character
indices
    font = ImageFont.truetype("Fonts/open_sans_bold.ttf", 35)
    font_index = ImageFont.truetype("Fonts/open_sans_bold.ttf", 11)

    # Create a draw object for drawing on the image
    draw_image = ImageDraw.Draw(txt)

    # Define the x-axis offsets for the three rows of characters on the keyboard
    row_offsets = [110, 135, 175]

    # Loop through all of the characters on the keyboard
    for i in range(30):
        # Calculate the row and column of the current character
        keyboard_row = i%3
        keyboard_column = math.floor(i/3)

        # Calculate the x and y coordinates of the character
        x = keyboard_column*58 + row_offsets[keyboard_row]
        y = keyboard_row*65 + 85

        # Calculate the coordinates for the character index and the character itself
        character_coords = (x, y)
        index_coords = (x-3, y-3)

        # Draw the character index and the character on the image
        draw_image.text(index_coords, str(i), font=font_index, fill=(0, 0, 0, 255))
        draw_image.text(character_coords, keyboard_characters[i], font=font, fill=(0,
0, 0, 255))

    # Create a composite image by blending the original keyboard image and the text
image
    out = Image.alpha_composite(start, txt)

    # Display the final image
    display(out)
```

Description:

This generates an image of a keyboard layout based on the best keyboard layout produced by the genetic algorithm. It does this by overlaying text on top of an image of a keyboard. The code first opens the keyboard image and converts it to RGBA format. It then creates a new image with a transparent background and loads the font for the keyboard characters and the font for the character indices. It then creates a draw object for drawing on the image. Next, it defines the x-axis offsets for the three rows of characters on the keyboard and loops through all of the characters on the keyboard. For each character, it calculates the row and column of the character, the x and y coordinates of the character, and the coordinates for the character index and the character itself. It then draws the character index and the character on the image using the draw object. Finally, the code creates a composite image by blending the original keyboard image and the text image, and displays the final image.

Output:

Figure 5 shows the SARMD layout which is the best optimised keyboard layout produced by the genetic algorithm after 300 generations.

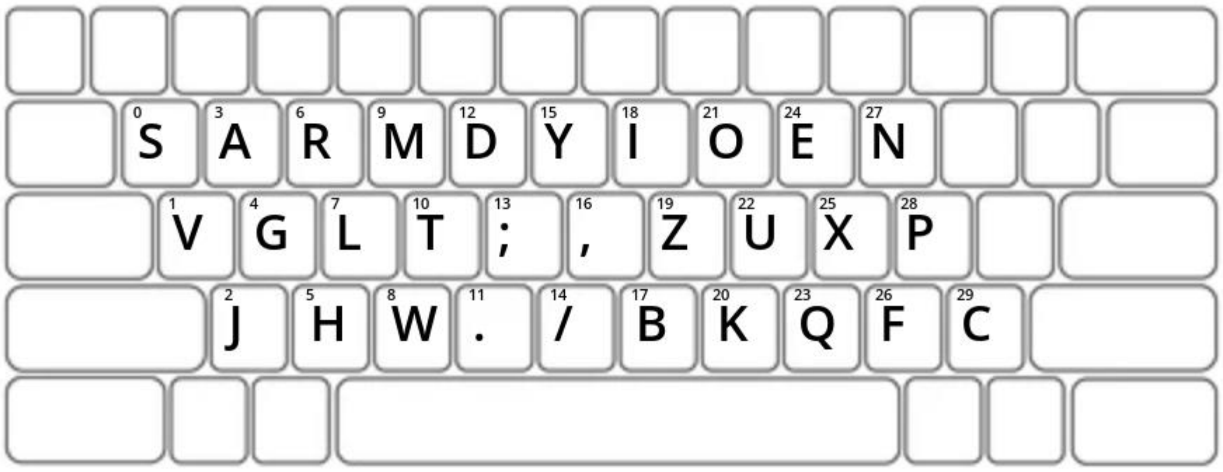

Figure 5

Observations

One can notice several things from the new outputted keyboard in Figure 5. For instance, the special symbols are in the middle of the keyboard, this makes sense since these keys are further than the starting keys than any other keys. All vowels are in the top row except for ‘U’ which is in the middle row. The consonants ‘RSTL’ are all in the top row and on the left side while ‘N’ is in the top row but on the right side. These five consonants (‘RSTLN’) are the most frequently used letters. Due to this I excepted them and all the vowels to be outputted in the middle row since they are used so frequently and since it is the starting row therefor the user will less distance to travel when typing.

Conclusion

The goal of this algorithm was to implement a genetic algorithm for the optimization of a more ergonomic and efficient keyboard layout for the standard English (United Kingdom) keyboard layout. The described approach produced a layout with good overall efficiency. As a result, the optimum layout discovered by this research is consistent with early predictions.

Figure 5 shows the final optimised keyboard. The genetic algorithm was run with a population size of 300 and for 300 generations. These values yielded the best result (Figure 5) since values higher than 300 the genetic algorithm was overfitting the dataset. Therefor the optimal values of 300 outputted the final keyboard, the SARMD keyboard.

The SARMD keyboard, which was the result of the genetic algorithm, was able to significantly reduce the distance travelled to type the text, making it a more efficient and effective keyboard layout than the QWERTY keyboard. This demonstrates the power of genetic algorithms in finding optimal solutions to complex problems, such as designing an efficient keyboard layout. After analysing the Distance travelled using the QWERTY Keyboard to type the dataset against the distance travelled when using the SARMD Keyboard, it was concluded that the SARMD Keyboard is around 45% more efficient.

It is impossible to establish that the proposed design is the best due to the numerous combinations of genetic algorithm's characteristics. It is possible that by making minor adjustments to the genetic algorithm configuration, alternative optimum outcomes can be obtained, which may favour one feature over another.
Individuals who have never used a keyboard before would benefit greatly from an optimized key-board arrangement. However, training would be required for individuals who already have some experience with the QWERTY layout. As a result, while the QWERTY layout is not the most efficient designated layout, experienced users are more adept at it than others.