DESIGNING A KEYBOARD USING GENETIC ALGORITHMS

Machine Learning: Classification, Search and Optimisation

Kelsey Bonnici
17203L

Table of Contents

Introduction

The QWERTY keyboard layout, which was named after the first six letters that appear in the top row, was designed in 1873 with the invention of the first commercial typewriter. The most popular theory surrounding the invention of this layout is that if a user pressed letters near each other too quickly, the machinery would jam. Thus, the resulting layout ensures that commonly typed letter pairings are not located next to each other [1]. With today's technology, we do not have this problem of the keys jamming but the QWERTY layout remains the most popular layout. Other keyboard layouts have emerged such as the DVORAK and COLEMAK layout which claimed to be more efficient, but by that point people had already gotten used to the QWERTY layout.

The scope of this project is to design a new and better keyboard layout. By reducing the distance that each finger has to travel, a user will take less time to write something. This will be done using a genetic algorithm. The algorithm will be trained on *The Picture of Dorian Gray* by Oscar Wilde[1] (located in book.txt). After filtering out invalid characters, the dataset comes to a final length of 424,081 characters.

Code

All the code for this project can be found in GAs.py. This code requires the following Python Modules to execute; random, numpy, math, csv.

printKeyboard()

The first function that can be found in the program is the printKeyboard() function. This function takes in a list called keyboard, which has the keyboard that will be printed out.

Through the use of for loops and if-else conditions, each list is printed out in the format of a keyboard as can be seen below. Note that instead of the whitespace character a '_' is printed for clarity.

```
Printing keyboard (using _ instead of whitespace for clarity)

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  ,  .  ?
          _
```

[1] Retrieved from https://gutenberg.org/

openDataset()

The openDataset() function is used to open and filter the dataset that will be used. This function takes in a list named keyboard which contains the keyboard and a string filename which contains the name of the file where the dataset is located as parameters.

The file is opened using the open() function and its contents are read and stored in the variable contents. While the contents of the file are being read, new line '\n' is being replaced by a whitespace and each character is turned into lowercase. The next step is for each character to be checked if it is valid. In this case, valid means that the character is in the given keyboard. If a character is valid it is appended to the string called text. Finally, the length of text is output to the user and the variable text is returned.

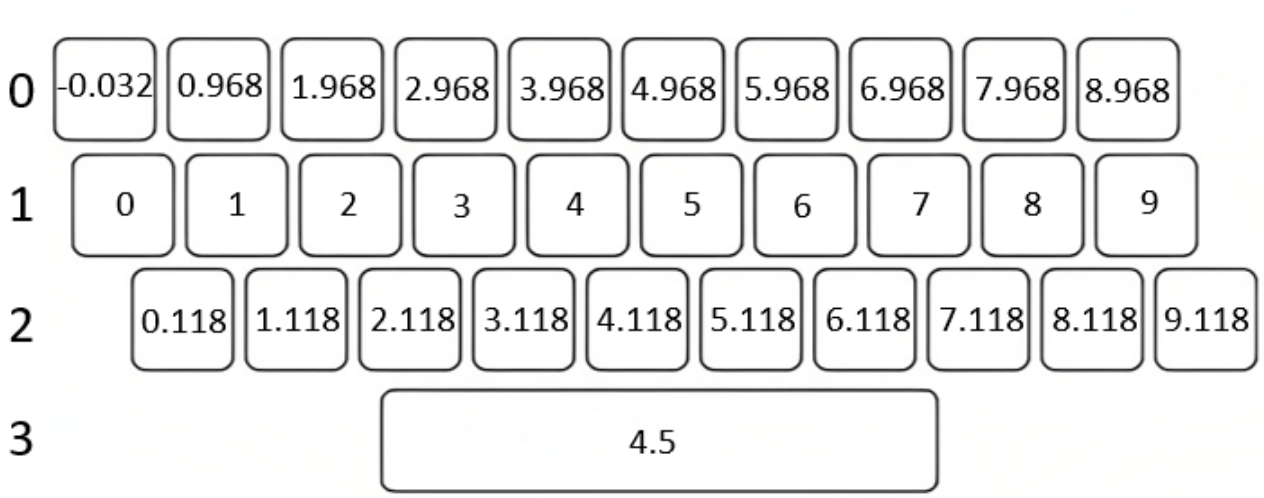
euclideanDistance()

The euclideanDistance() function is used to calculate the Euclidean distance between two coordinates. As parameters, this function takes in two coordinates, coordinate1 and coordinate2, in the form of tuples. The difference between these two coordinates is found by turning them into a numpy array and subtracting them from each other. The function then returns the norm which is calculated using the norm function from the numpy module.

distanceMoved()

The distanceMoved() function is used to calculate the total distance that each finger has to move to write a string. The function takes in two parameters, a keyboard and the text.

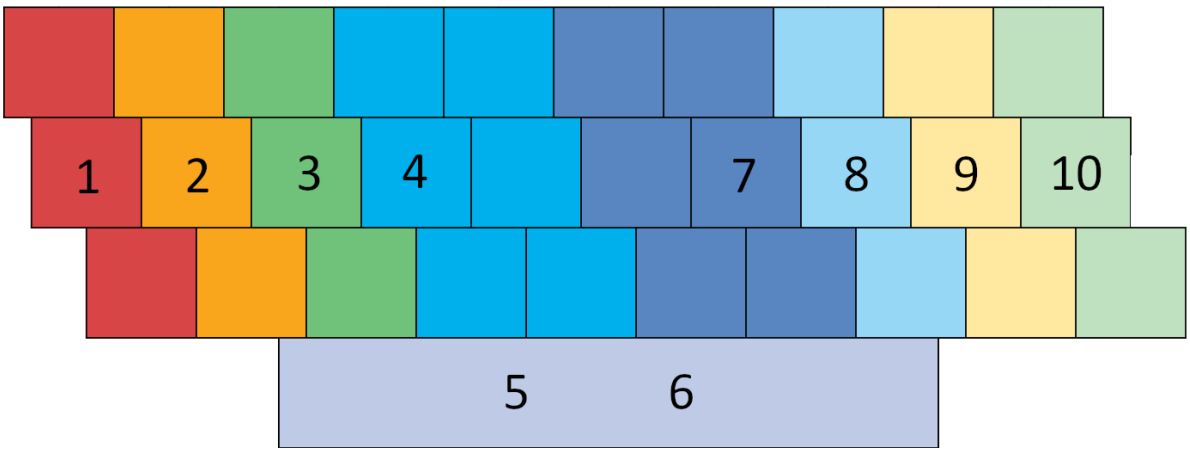
The dictionary keyCoordinates contains the index of a key and its location on the keyboard. The diagram below allows for better visualisation of how to coordinates were plotted.



The layout of a standard keyboard is not a grid. There is an offset difference between each row. The second row is given the value 0. Moving one row up, the key above is offset by -0.032. Moving one row down from the second, the key below it is offset by 0.118. Moving to a key left or right has a distance of 1. Since the space bar is located in the middle of the

layout it has a value of 4.5 since 9 divided by 2 is equal to 4.5. The values of 0.032 and 0.118 were both taken from the video linked in the assignment.

The position of each finger is stored in a dictionary called fingers. The keys where each finger is allowed to move is stored in a dictionary called fingerAllowed. The values found in these two dictionaries can be seen demonstrated in the diagram below.



Each finger starting from the little finger on the left hand and ending on the little finger on the right hand has its default position set as seen in the diagram above. Each finger is allowed to move to a key with the same colour. Since fingers 5 and 6, which represent both thumbs, are always in the same position together, finger 6 has been omitted from the list.

The function enters a for loop to iterate over the text that was given in as a parameter. The coordinate of the current letter is retrieved according to its index in the keyboard list. If the coordinate that was retrieved is not found in the fingers list (i.e. there is not a finger currently on that key), the index of the key that needs to be pressed is retrieved. In the case that the coordinate is found in the list, it means that the finger does not have to move so nothing is added to the distanceMoved variable. After that, the finger which is allowed to move to that key is also retrieved. The coordinates of the key where the finger is currently located and where it needs to moved are retrieved. The Euclidean distance between those two keys is then calculated using the previously mentioned euclideanDistance() function and it is added to the distanceMoved variable which stores the total distance that has been moved. After the calculation is done, the fingers new coordinate is set. This means that when a finger moves to press a key, it does not move back to its original position.

After iterating over every letter in text, the function returns the distanceMoved.

createFirstPopulation()

This function is used to create the first population that will be used. It takes in the size of the population and a keyboard as the parameter. The keyboard here must be the same one that was used in the openDataset() function or the program will produce errors.

The variable population is a list of lists, where each list is a keyboard. The keyboards are randomised using the sample() function from the random module. The function then returns the population variable.

sortPopulation()

This function is used to sort a population. It takes in as parameters the population that will be sorted and the text.

The population list is sorted using the in-built sorted() function. The keyboards inside the population list are sorted in ascending order based on the total distance moved which is calculated using the distanceMoved() function.

The function then returns the variable sorted_population.

generationsLoop()

This function is used to create and iterate through each generation. As parameters, this function takes in:

- population – the first population
- text – the dataset
- iterations – the number of generations that will be created
- elitism – the percentage of chromosomes to be put in the next generation
- mutate – the probability of a mutation happening
- mutateChoice – an integer representing which mutation type is to be done
- crossoverChoice – an integer representing which crossover type is to be done

The function starts off by entering a loop that goes on for as long as was specified in the iterations variable. The function prints out the number of the generation it is working on to the user. The population is sorted using the sortPopulation() function which was explained above and stored into the variable sorted_population. The population variable is reinitialised and chromosomes from the sorted_population are put into it according to the percentage of elitism (if elitism happens to be 0, then nothing gets added to the population).

The function then enters another for loop that will iterate to fill the rest of the population. Two parents are chosen at random from the top 50% of sorted_population. These parents can be the same and can also be found in the new population if there is an elitism percentage bigger than 0.

The function then moves on to create the child of the two parents. The crossover function that is used is determined by the crossoverChoice variable that was specified as a parameter. The function then moves on to the mutation part. Here the function generates a random number between 0 and 1. If this number happens to be less than the mutate variable that was specified as a parameter, the child is mutated. The type of mutation that is

performed is determined by the mutateChoice variable that was specified. After the child is created and possibly mutated, it is appended to the population.

After all the generations have been created and iterated through, the final population is sorted and returned by the function.

Note that a generationsLoopFile() function can be found in the code. This function works the same way as the generationsLoop() function but it also logs the best distance of every generation to a csv file. This was done to be able to have a record of all the distances and generate the graphs for the evaluation.

singlePointCrossover()

This function is used to perform a single point crossover over two parents. The function takes in two parents as a parameter.

The function starts off by generating the list where the child will be stored. It also generates a random number between 1 and 29 which will be used as the crossover point.

To demonstrate how the function works, a dry run with two random keyboard layouts will be done. The green cells indicate characters that were copied into the child, the yellow cells indicate characters that had been duplicate. In this run, the crossover point was 5.

In the first for loop, which goes on according to the crossover point (in this case 5 times), each letter from parent1 is copied into the child.

Index	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	30
parent1	,	f	e	h	g	d	n	a	t	w	b	i	q	l	v	;	k	y	u	m	c	z	p	o	x	j	_	?	s	.	r
parent2	s	b	g	y	d	;	m	c	f	l	a	h	i	,	_	e	z	.	q	u	o	j	r	t	x	v	w	k	n	p	?
child	,	f	e	h	g																										

Next, the function enters a while loop to keep iterating while a '-' still appears in the child list. Here the second half of parent2 is copied into the child, while also taking care to not have duplicate letters.

Index	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	30
parent1	,	f	e	h	g	d	n	a	t	w	b	i	q	l	v	;	k	y	u	m	c	z	p	o	x	j	_	?	s	.	r
parent2	s	b	g	y	d	;	m	c	f	l	a	h	i	,	_	e	z	.	q	u	o	j	r	t	x	v	w	k	n	p	?
child	,	f	e	h	g	;	m	c	l	a	i	_	z	.	q	u	o	j	r	t	x	v	w	k	n	p	?				

In the case above, the loop has reached the end of parent2, but the child is still not complete. This is due to the duplicate letters (indicated by yellow). Here, the function will start iterating over parent2 from the beginning and the letters will be taken from there.

Index	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	30
parent1	,	f	e	h	g	d	n	a	t	w	b	i	q	l	v	;	k	y	u	m	c	z	p	o	x	j	_	?	s	.	r
parent2	s	b	g	y	d	;	m	c	f	l	a	h	i	,	_	e	z	.	q	u	o	j	r	t	x	v	w	k	n	p	?
child	,	f	e	h	g	;	m	c	l	a	i	_	z	.	q	u	o	j	r	t	x	v	w	k	n	p	?	s	b	y	d

Since both parents are always randomly generated from the same keyboard, there is no need to check that all the letters are included in the child. At the end the function returns the child keyboard.

twoPointCrossover()

This function is used to perform a two-point crossover over two parents. The function takes in two parents as a parameter.

The function starts in a similar manner to the single point crossover where it first generates the list where the child will be stored. Then it generates two random numbers to be used as crossover points.

The way the function works will be demonstrated using a dry run as was done for the single point crossover. In this case, the two crossover points were 5 and 26.

For the first part, it is done in the same way as the single point crossover, where the characters are copied into the child up until the crossover point.

Index	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	30
parent1	q	l	i	a	_	s	y	w	h	m	n	;	j	.	z	f	d	o	x	g	t	k	u	?	v	c	p	,	r	b	e
parent2	m	;	i	p	n	t	o	e	f	l	y	b	a	r	s	,	v	h	d	x	w	_	c	.	g	u	q	k	z	j	?
child	q	l	i	a	_																										

For the second part, it is also done in the same way as the single point crossover, but the while loop condition is slightly different. Instead of it going until there aren't any '-' in the child list, it goes on until it hits the second crossover point.

Index	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	30
parent1	q	l	i	a	_	s	y	w	h	m	n	;	j	.	z	f	d	o	x	g	t	k	u	?	v	c	p	,	r	b	e
parent2	m	;	i	p	n	t	o	e	f	l	y	b	a	r	s	,	v	h	d	x	w	_	c	.	g	u	q	k	z	j	?
child	q	l	i	a	_	t	o	e	f	y	b	r	s	,	v	h	d	x	w	c	.	g	u	k	z	j					

As can be seen in the image above, the characters 'k', 'z' and 'j' are still included in the child even if they are past the crossover point. This is because the function compares the index of the child list to the crossover point. The reason was done this way is because of duplicate characters.

For the third part, the function starts iterating over the first parent and adding any character which has not been added yet to the child. Since the two keyboards have been generated from the same keyboard and are of the same length, if there are (in this example) 5 letters yet to be added to complete the child list, those will be the only character in parent1 which will not be found in the child list. This guarantees that the generated child has no duplicates and contains each character.

Index	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	30
parent1	q	l	i	a	_	s	y	w	h	m	n	;	j	.	z	f	d	o	x	g	t	k	u	?	v	c	p	,	r	b	e
parent2	m	;	i	p	n	t	o	e	f	l	y	b	a	r	s	,	v	h	d	x	w	_	c	.	g	u	q	k	z	j	?
child	q	l	i	a	_	t	o	e	f	y	b	r	s	,	v	h	d	x	w	c	.	g	u	k	z	j	m	n	;	?	p

At the end, the function returns the child that was generated.

swap()

The swap() function is one of the three mutation functions in this code. The function takes in a keyboard as a parameter.

The function starts off by generating two random numbers which will be used as the indexes of the keys that will be swapped. There is also a check to make sure that the two numbers that have been generated are not the same.

The characters in the positions are then swapped around and the keyboard is returned.

Below is a dry run of the function with the values that were affected in blue. In this case the two random numbers were 8 and 16.

Index	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	30
Before	;	f	u	k	d	y	i	s	z	j	h	v	p	n	w	e	,	q	m	?	b	g	a	r	.	c	o	x	_	t	l
After	;	f	u	k	d	y	i	s	,	j	h	v	p	n	w	e	z	q	m	?	b	g	a	r	.	c	o	x	_	t	l

scramble()

The scramble() function is the second mutation function to appear. This function also takes in a keyboard as a parameter

The functions start off by generating two random numbers in the same manner as the swap() function. There is also an extra check to make sure that random1 is always smaller than random2.

The section from index random1 to index random2 in the keyboard in copied into a list named copy. It is then shuffled using the shuffle from the random module. The section in the original keyboard is the replaced by the newly shuffled part and the keyboard is returned.

Below is a dry run of the function with the characters that were affected in blue. In this case the two random numbers were 17 and 29.

Index	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	30
Before	m	j	b	i	;	l	q	r	,	e	t	y	n	h	d	.	u	z	j	c	w	a	g	v	x	p	o	k	f	_	s
After	m	j	b	i	;	l	q	r	,	e	t	y	n	h	d	.	u	v	c	g	z	k	o	j	f	p	a	x	w	_	s

invert()

The invert() function is the third and last mutation function to appear. Like the previous functions, this also takes in a keyboard as a parameter.

The function works the same as the scramble() function where it takes a section of the keyboard, but instead of shuffling it, it is reversed using the in-built reversed() function. The keyboard is then returned.

Below is a dry run of the function with the characters that were affected in blue. In this case, the two random numbers were 14 and 28.

Index	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	30
Before	a	p	w	g	l	h	o	d	_	b	q	.	r	s	,	j	c	z	e	x	k	;	m	y	n	u	v	?	f	i	t
After	a	p	w	g	l	h	o	d	_	b	q	.	r	s	?	v	u	n	y	m	;	k	x	e	z	c	j	,	f	i	t

testCases()

This function contains all the test cases that will be discussed later on in the evaluation. It contains each function run with the respective parameters for each test. It uses the generationsLoopFile() function so all the data is logged and also created a .txt file with the final keyboard.

Note that running this function will take a considerable amount of time and overwrite any files with the same names as declared in the tests.

Main

Upon executing the program, the user will greeted with a menu with three options as can be seen below.

```
Designing a keyboard using genetic alogrithms

Choose an option
1. Use default values
2. Enter custom values
3. Test cases (will take a considerable amount of time to execute)
```

Option 1 will execute the algorithm using these values:

- Population size: 100
- Number of generations: 100
- Elitism: 10%
- Probability of mutation: 40%
- Mutation type: Scramble
- Crossover type: Two Point Crossover

The default values were decided after the evaluation was done. Explanation for why these values were chosen will be given later on.

Choosing option 2 will allow the user to modify each value to their liking.

Option 3 will run the testCases() function. As stated before, this function takes a considerable amount of time to execute and will overwrite files with the same names.

Evaluation

The evaluation was done over 11 test cases to investigate the effect of changing four variables: the population size, mutation rate, elitism percentage and the type of mutation. With both single point crossover and two-point crossover, there is a total of 22 test cases.

All the test cases start from these variables and they are changed accordingly. These values are:

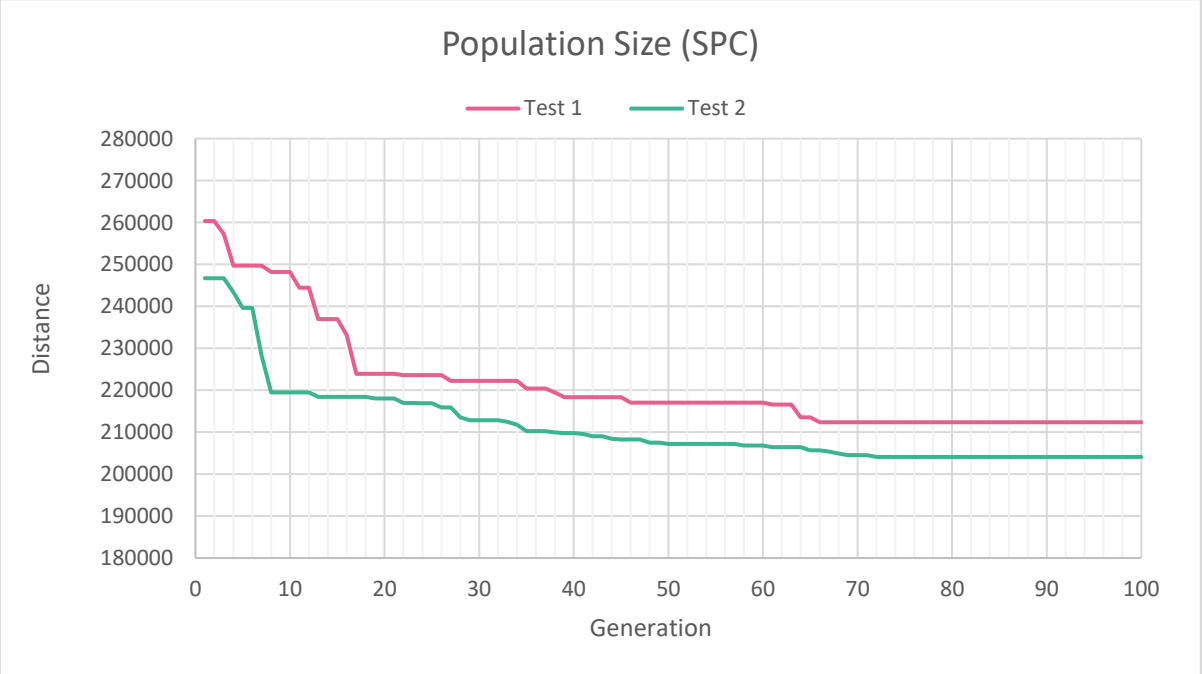
- Population Size: 100
- Number of generations: 100
- Elitism rate: 10%
- Mutation probability: 40%
- Mutation type: Scramble

Population Size

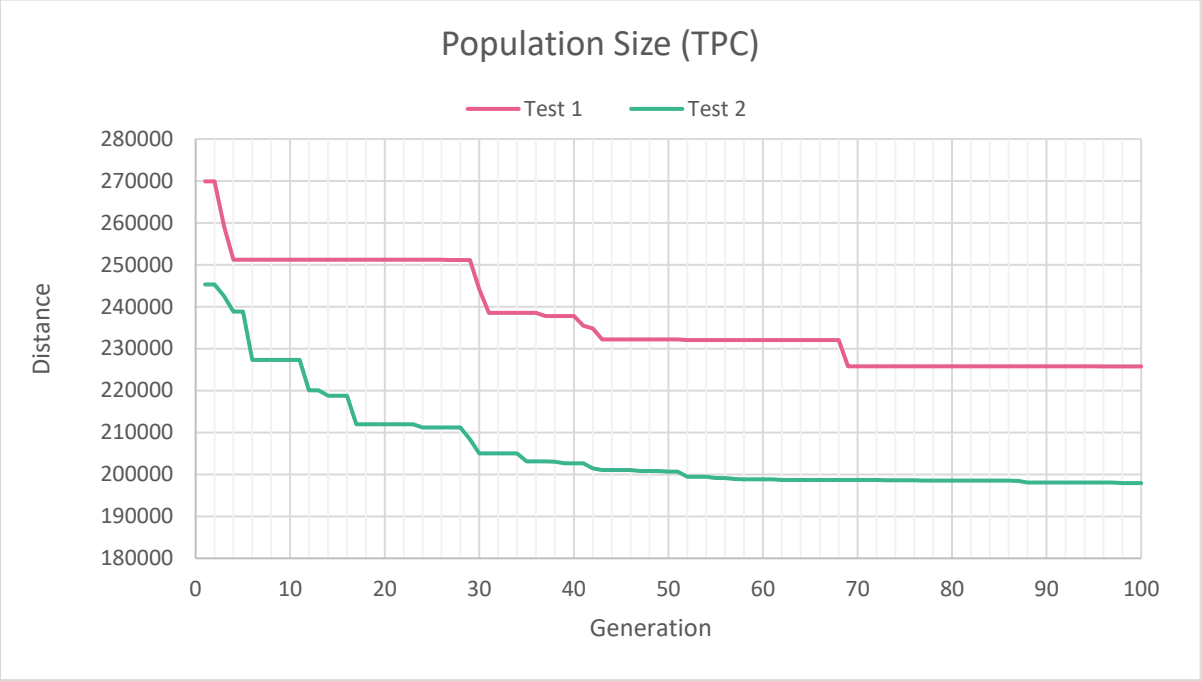
To investigate the effect of population size, two tests were done. One with a population of 10 and the other of a 100.

	Test 1	Test 2
Population	10	100
Generations	10	10
Elitism	10%	10%
Mutation	40%	40%

Single Point Crossover



Two-Point Crossover



The two graphs gave similar results of the second test with a higher population giving a lower final distance. But one problem with these graphs is that it could be argued that this difference (especially in graph 1) correlates with the difference in the starting point of the distance. As the function currently is, the first population that is created is completely random which can result in there being a big discrepancy in the starting distances. This problem could have been avoided by turning it into a heuristic initialisation by including the QWERTY layout in the first population. As a general rule in genetic algorithms heuristic initialisation is not recommended [2]. In this case, it would have ended up that every keyboard is a variation of the QWERTY keyboard instead of a completely new layout.

Another problem with this test is that when increasing the population, the time it takes for the whole program to execute also increases due to the limitation of the machine this task was done on. Adding a third test with a higher population might have also helped with the previously mentioned problem because there would be another graph to compare to.

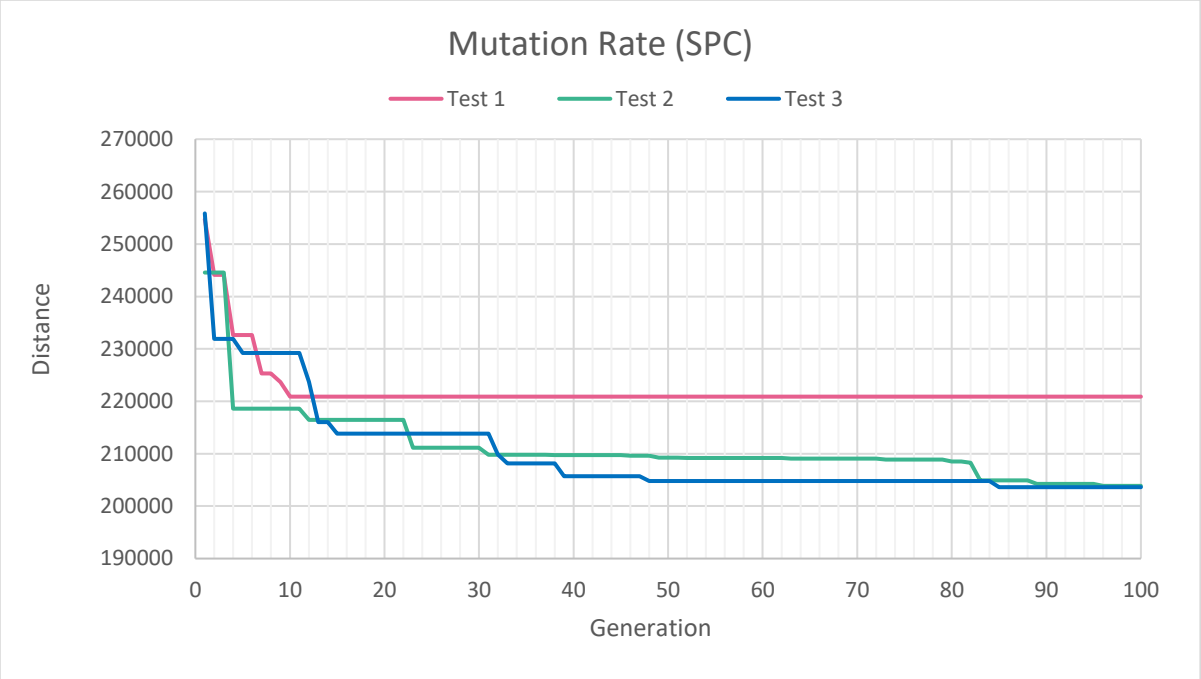
A good genetic algorithm needs a population that is large enough to have enough diversity but not too large as it will slow down [2]. A population of 100 was sufficient enough and this was used as a default value for every test.

Mutation Probability

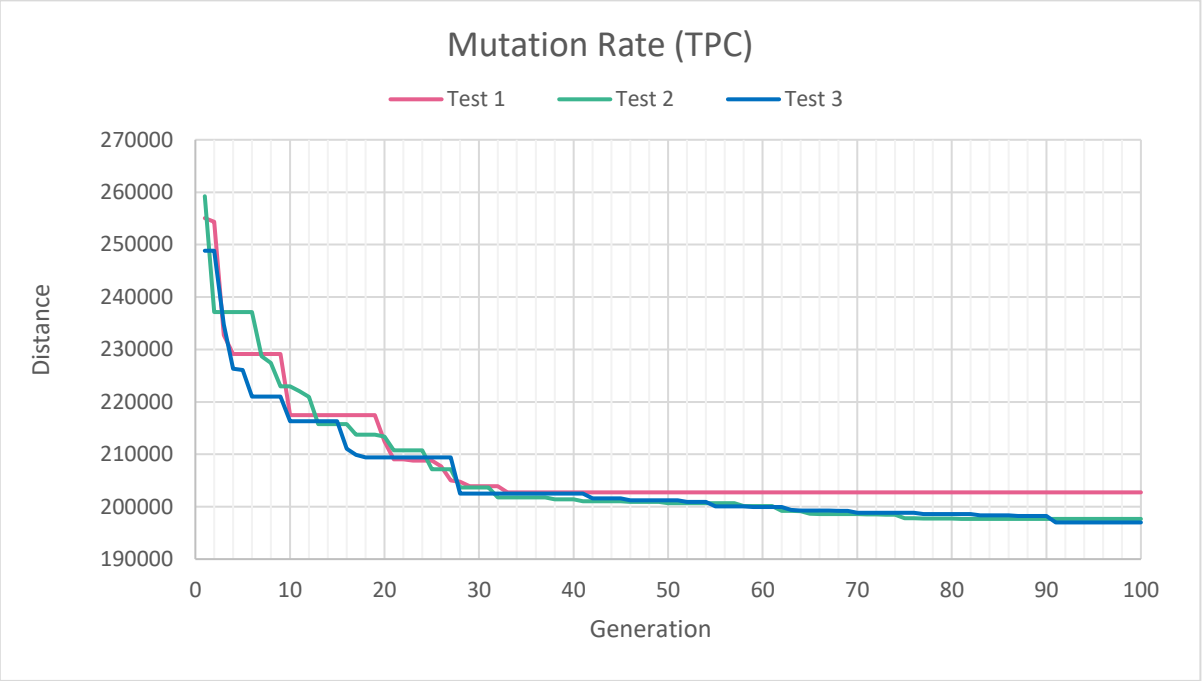
To investigate the effect of mutation probability, three tests were done with varying probability of mutation.

	Test 1	Test 2	Test 3
Population	100	100	100
Generations	100	100	100
Elitism	10%	10%	10%
Mutation	0%	40%	100%

Single Point Crossover



Two Point Crossover



In both cases, it is clear that having no mutation (test 1) gives substandard results. It is important to have some sort of mutation in a genetic algorithm as this allows for diversity to be maintained and stops early convergence [3]. For TPC, test 1 gave a final distance that is closer to the convergence point. This is due to that crossover method introducing more diversity by design. This will be discussed more in detail later on.

For test 2 and test 3 in both graphs there isn't much of a difference between the results. But as a general rule, it is not recommended to have a mutation probability that is very high because the GA will be reduced to a random search [3]. In test 3, every child that is generated is mutated. One reason why the difference between the final distances might be so low is that the scramble mutation takes random sized sections to mutate and does not have a set length. Since that data was not recorded, it is difficult to say whether that is the case or not.

The reason why the scramble was chosen as the mutation type was that it gave the most different result out of the three types of mutations. The effect of different types of mutations will be discussed more in detail later.

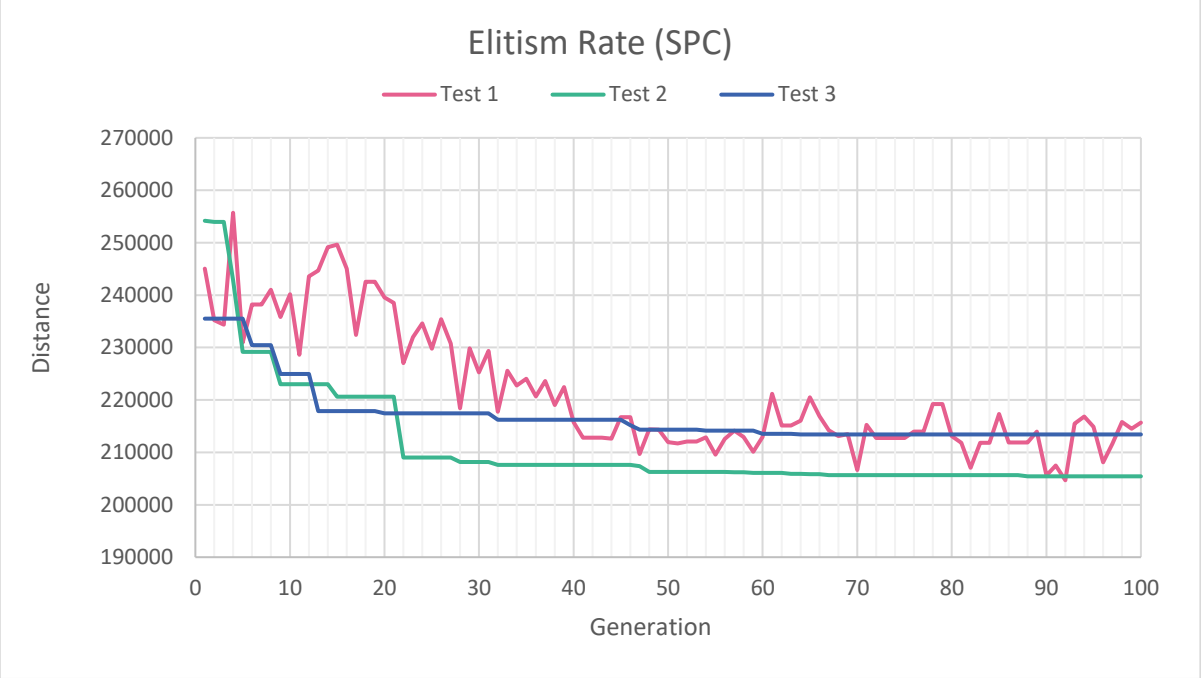
With this reasoning, the default value that was chosen for the mutation rate was 40%.

Elitism

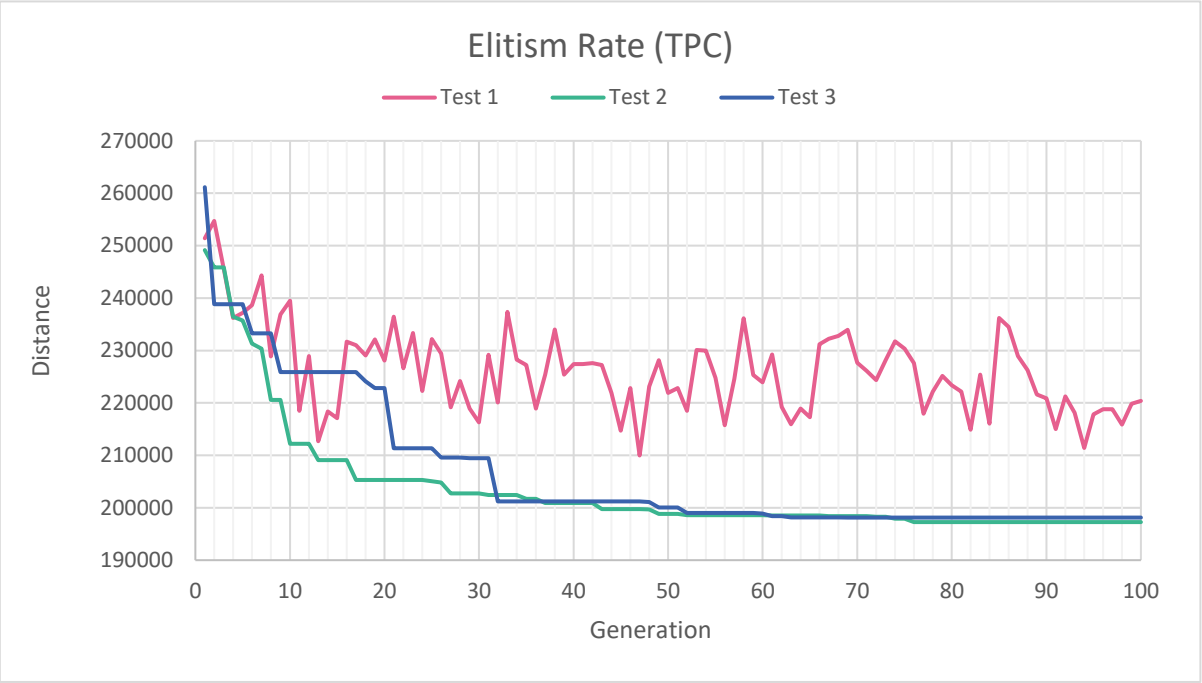
To investigate the effects of elitism, three tests were done with varying rates of elitism.

	Test 1	Test 2	Test 3
Population	100	100	100
Generations	100	1001	100
Elitism	0%	10%	50%
Mutation	40%	40%	40%

Single Point Crossover



Two Point Crossover



For both crossover types, it is clear that having no elitism (test 1) gives substandard results. In some cases, the best keyboard in the new population has a higher distance than the one before it. This does not allow the genetic algorithm to converge.

Having an elitism that is too high can hurt diversity in the population because the same few chromosomes are being kept. In the case of test 3, the elitism of 50% meant that every chromosome that was kept in that next population also had a chance of being a parent. Having the elitism rate be this way gave substandard results as seen in the first graph for SPC. This was not the case for the second graph but this is due to TPC introducing more diversity by design.

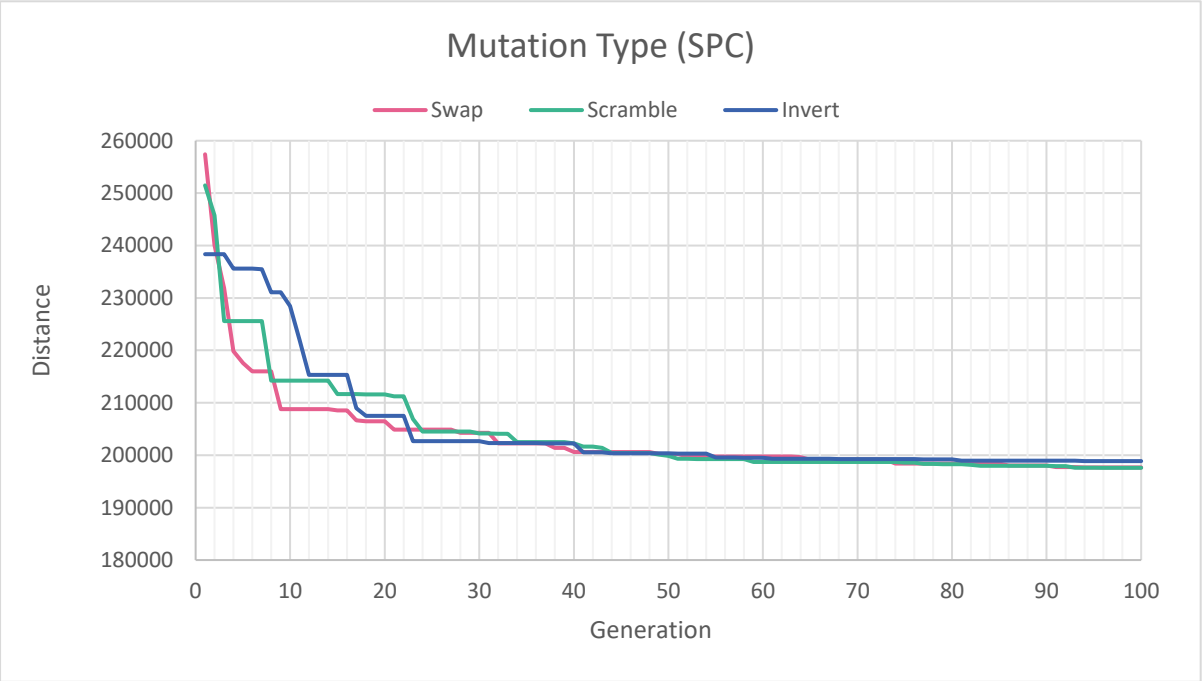
It is clear (especially in the first graph) that having a lower elitism rate leads the algorithm closer to a convergence point. For this reason the elitism rate of 10% was chosen as the default value.

Mutation Type

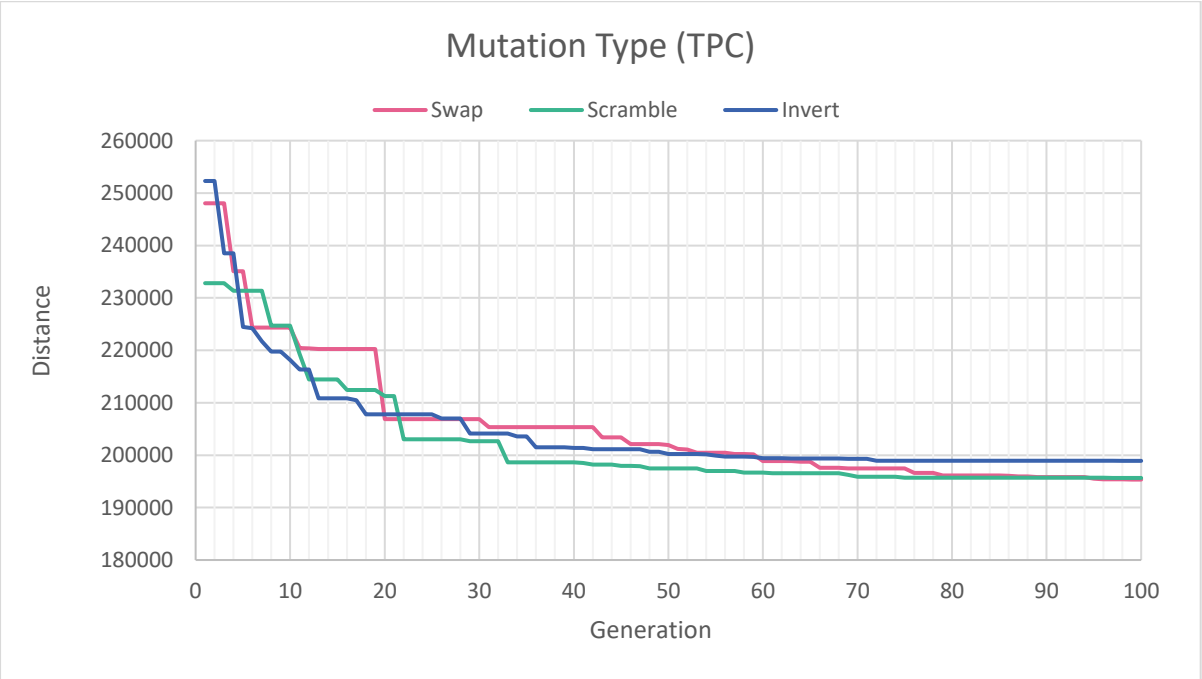
To investigate the effects of the different mutation types, a test was performed on each type with the following values

Population	100
Generations	100
Elitism	10%
Mutation	40%

Single Point Crossover



Two Point Crossover



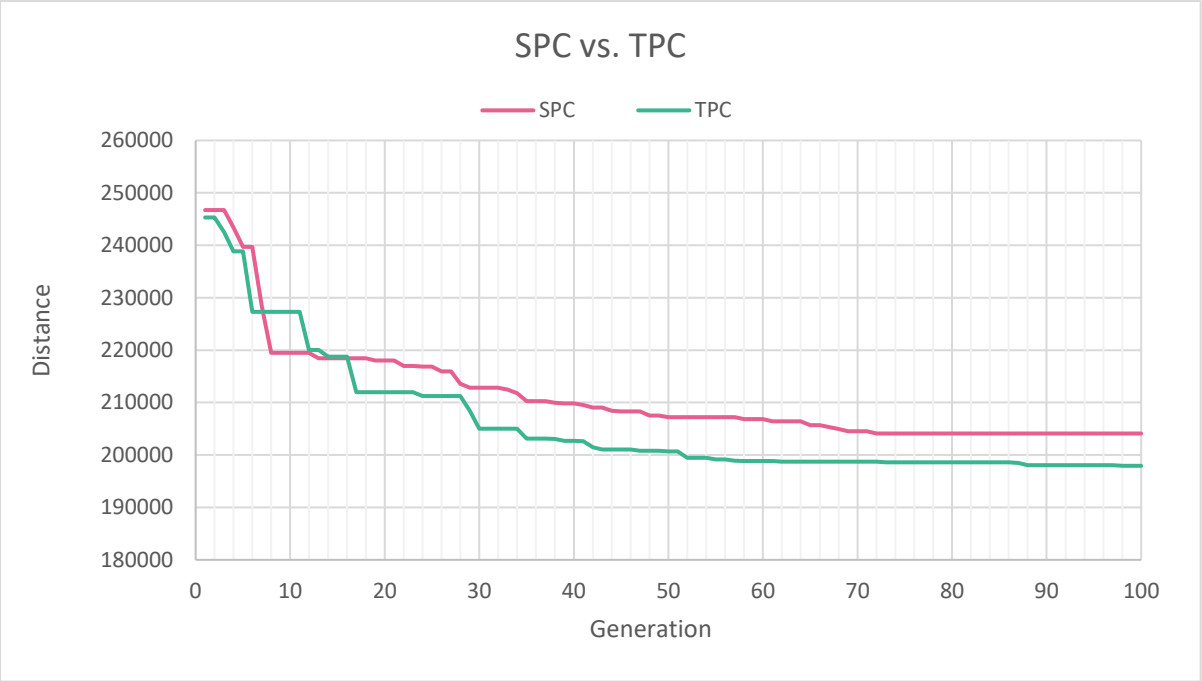
From these results, it is not clear whether the type of mutation that is done makes a difference to the final results. All graphs converged towards one point even with having drastically different starting points.

It was expected that scramble and invert would outperform swap by a lot since swap does such a minor change to the child but this was not the case. The reason this did not happen could be due to the section of the chromosome that is affected is randomly chosen and

does not have a set length. Another reason could be that since the probability is set a 40%, the number of mutations that happened in each GA is not the same.

Single Point Crossover vs. Two-Point Crossover

To compare the two crossover methods, the same data that was used for the two tests labelled test 2 that was used for the population size comparison will be used. This is due to those tests having the values which were set as the default values.



As can be seen clearly from the graph, the two-point crossover performed better than the single point crossover. This is due to the TPC introducing more diversity into the population by its design [4].

Conclusion

In conclusion, using all the data that was gathered, the following parameters were deemed the best to give a more efficient keyboard layout:

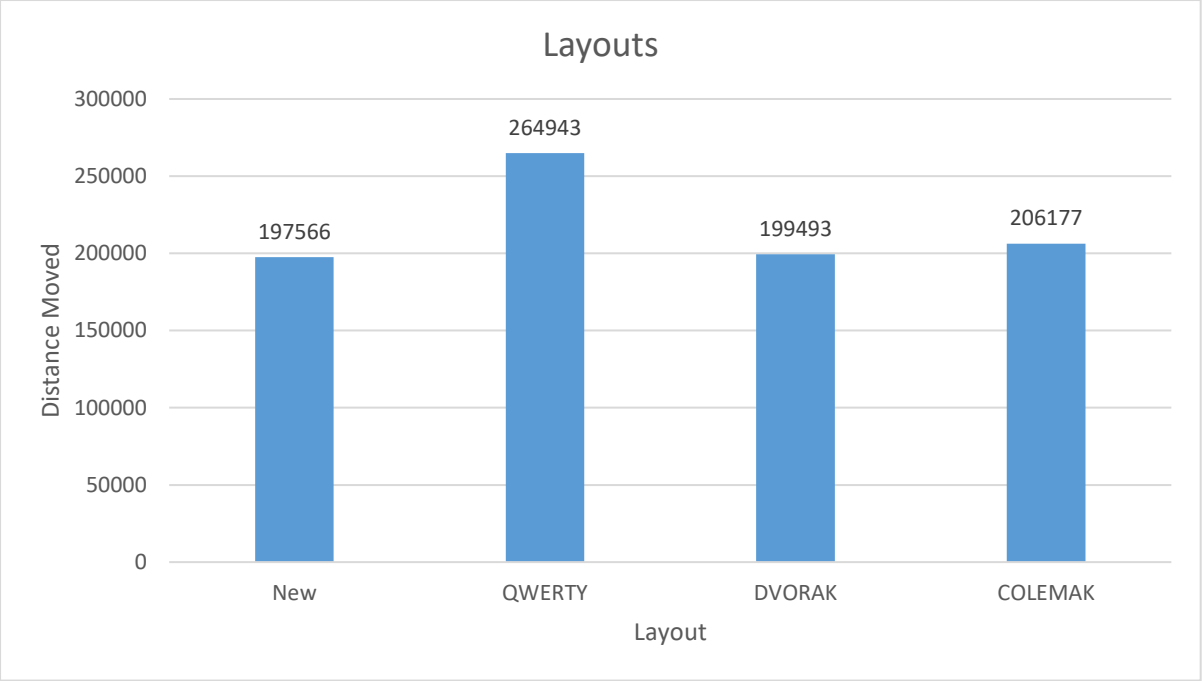
- Population size: 100
- Number of generations: 100
- Elitism: 10 %
- Probability of mutation: 40 %
- Mutation type: Scramble
- Crossover Type: Two Point Crossover

Running the algorithm once with these parameters gave the following keyboard with a distance of 197,566.

```
;  r  e  k  w  f  m  d  x  .
c  n  y  u  t  l  h  s  _  o
a  j  z  b  g  v  p  ?  q  ,
            i
```

It can be observed that some of the more commonly used letters are found in the middle row and where the spacebar would usually be located. This is because those keys require the least distance to be moved since by default, the fingers are located on those keys.

Below is a comparison of the QWERTY, DVORAK, COLEMAK and the new keyboard layout.



The newly generated keyboard outperforms the other layouts. It only slightly outperforms the DVORAK layout with less than 1% decrease. The biggest decrease is from the QWERTY layout with a 25% decrease in the distance moved.

References

[1] J. Stamp, “Fact or Fiction? The Legend of the QWERTY Keyboard,” Smithsonian Magazine, 3 May 2013. [Online]. Available: https://www.smithsonianmag.com/arts-culture/fact-of-fiction-the-legend-of-the-qwerty-keyboard-49863249/. [Accessed 26 December 2022].

[2] tutorialspoint, “Genetic Algorithms - Population,” tutorialspoint, [Online]. Available: https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_population.htm. [Accessed 31 December 2022].

[3] tutorialspoint, “Genetic Algorithms - Mutation,” tutorialspoint, [Online]. Available: https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_mutation.htm. [Accessed 31 December 2022].

[4] S. P. Dubey, K. N Gopalakrishna, S. Balaji and M. Sathish Kumar, “A Comparative Study on Single and Multiple Point Crossovers in a Genetic Algorithm for Coarse Protein Modeling,” *Critical Reviews™ in Biomedical Engineering,* vol. 46, no. 2, pp. 163-171, 2018.

Statement of Completion

Item	Completed
Implement ‘base’ genetic algorithm	Yes
Two-point Crossover	Yes
Implemented a mutation operation	Yes
Elitism	Yes
Good evaluation and discussion	Yes

Plagiarism Declaration Form

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

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

I, the undersigned, declare that the assignment submitted is my work, except where acknowledged and referenced.

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

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

Kelsey Bonnici _____ _____

Student Name Signature

ICS2207 Designing a keyboard using genetic algorithms

Course Code Title of work submitted

02/01/2023

Date