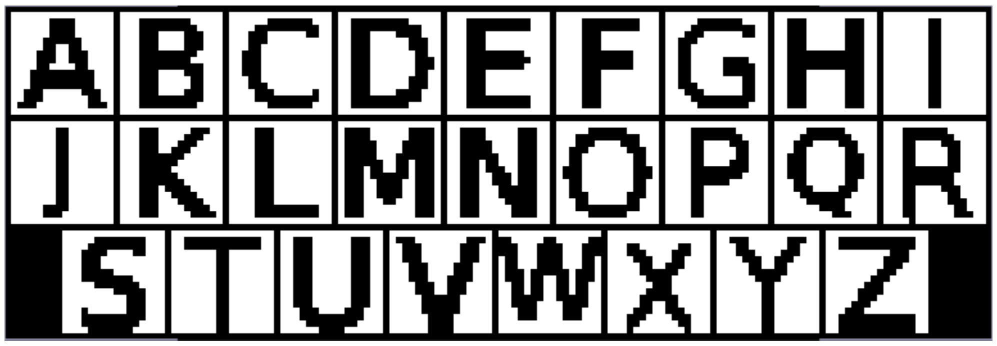
MANUEL OLIAS LOPEZ

Optical Character Recognition

Optical Character Recognition (OCR) is a technology used to convert images of text into machine-readable text, which can then be edited, searched, and analyzed digitally. In this report, we present a genetic algorithm for solving an OCR problem involving alphabet characters represented as 16x16 grayscale images. The goal of the problem is to find the position of a few pixels that distinguish the characters of the alphabet from each other. My genetic algorithm is designed to find the optimal combination of pixels that yields the highest accuracy in character recognition.

First, I will try to solve the problem for the Latin alphabet, it contains 26 characters. In every image each pixel can take only two values, black or white (0 or 1). Therefore, theoretically, there could exist 5 pixels that gives us enough information to tell each character from one another. That is because $2^5 = 32 > 26$. However, $2^4 = 16 < 26$, then the best possible solution for the problem has at least 5 pixels. The images used in this first case are the following:



For the genetic algorithm, I had to define a couple of functions:

INITIAL POPULATION

I set the population size to 100 individuals. Each individual consists of a random list of 5 numbers, without repetition $[n_0, n_1, n_2, n_3, n_4]$, $0 <= n_i < 256$ for any i = 0,1,2,3,4.

Each number represents a pixel in the 16x16 grid following this rule:

n -> (n%16, floor(n/16)) (E.g. 123 -> 11th row, 7th column)

Therefore, each individual represents a set of 5 pixels in the grid.

FITNESS FUNCTION

This function assigns to each individual a number. A higher number represents a better individual. Given an individual, I will compute a list of 5 values for each letter. These values are the result of looking at the 5 pixels at the given image of the letter, they are either 0 or 1 (black or white). Now, by looking at all the lists, if there is any list repeated it means there is uncertainty between two letters. I will subtract one to the fitness for every repeated list.

Therefore, my goal is to find an individual with a 0 fitness.

PARENT SELECTION METHOD

After trying Roulette-wheel and Tournament selection methods, I found the latter provides better results. To select the parents for the following iterations I sorted all individuals of the current generation and only kept the best 50 (50%) as the set of parents.

CROSSOVER METHOD

After taking two random individuals from the set of parents, I generated two children from randomly picking a position from which I performed the crossover.

E.g. Having parents [23, 45, 67, 189, 234], [34, 57, 93, 134, 197] and choosing 3 as the crossover position we obtain children [23, 45, 67, 134, 197] and [34, 57, 93, 189, 234].

I will repeat this process until obtaining 100 children that, after the proper mutations, form the following generation.

MUTATION METHOD

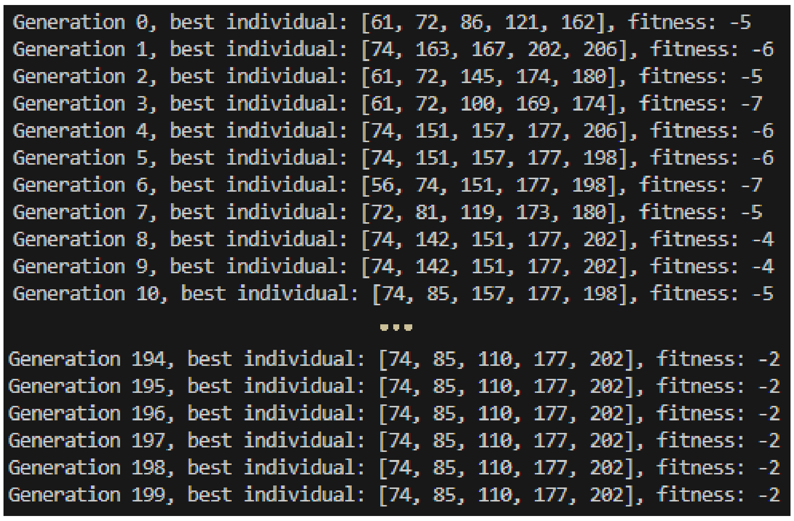
For this method, a mutation rate needs to be defined. In this case I found 0.05 works the best. For every individual of the new generation and for every one of its pixels, there is a 0.05 chance that they mutate. If a pixel mutates, it will just transform in another randomly selected pixel of the grid.

E.g. Having individual [23, 45, 67, 189, 234] and having mutated its 3rd may result into

[23, 45, 189, 204, 234].

WEIGHT MAP

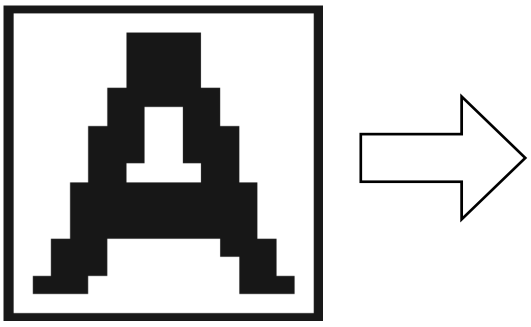
With all that, after many iterations I couldn't find any solution for the problem. The best iteration I found had a fitness of -2 and all the iterations looked like this:

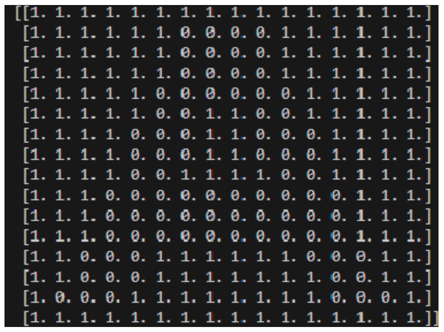


```
Generation 0, best individual: [61, 72, 86, 121, 162], fitness: -5
Generation 1, best individual: [74, 163, 167, 202, 206], fitness: -6
Generation 2, best individual: [61, 72, 145, 174, 180], fitness: -5
Generation 3, best individual: [61, 72, 100, 169, 174], fitness: -7
Generation 4, best individual: [74, 151, 157, 177, 206], fitness: -6
Generation 5, best individual: [74, 151, 157, 177, 198], fitness: -6
Generation 6, best individual: [56, 74, 151, 177, 198], fitness: -7
Generation 7, best individual: [72, 81, 119, 173, 180], fitness: -5
Generation 8, best individual: [74, 142, 151, 177, 202], fitness: -4
Generation 9, best individual: [74, 142, 151, 177, 202], fitness: -4
Generation 10, best individual: [74, 85, 157, 177, 198], fitness: -5
                              ...
Generation 194, best individual: [74, 85, 110, 177, 202], fitness: -2
Generation 195, best individual: [74, 85, 110, 177, 202], fitness: -2
Generation 196, best individual: [74, 85, 110, 177, 202], fitness: -2
Generation 197, best individual: [74, 85, 110, 177, 202], fitness: -2
Generation 198, best individual: [74, 85, 110, 177, 202], fitness: -2
Generation 199, best individual: [74, 85, 110, 177, 202], fitness: -2
```

After not finding any solution after many iterations I decided to modify the fitness function and use a weight map to obtain more information from each pixel.

To work with the images, I first needed to convert it to a more practical format, a matrix:



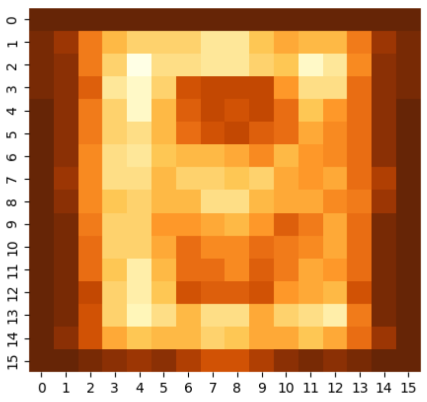


```
[[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 0. 0. 1. 1. 0. 0. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 0. 0. 0. 1. 1. 0. 0. 0. 1. 1. 1. 1.]
 [1. 1. 1. 1. 0. 0. 0. 1. 1. 0. 0. 0. 1. 1. 1. 1.]
 [1. 1. 1. 1. 0. 0. 1. 1. 1. 1. 0. 0. 1. 1. 1. 1.]
 [1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1.]
 [1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1.]
 [1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1.]
 [1. 1. 0. 0. 0. 1. 1. 1. 1. 1. 1. 0. 0. 0. 1. 1.]
 [1. 1. 0. 0. 0. 1. 1. 1. 1. 1. 1. 1. 0. 0. 1. 1.]
 [1. 0. 0. 0. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]]
```

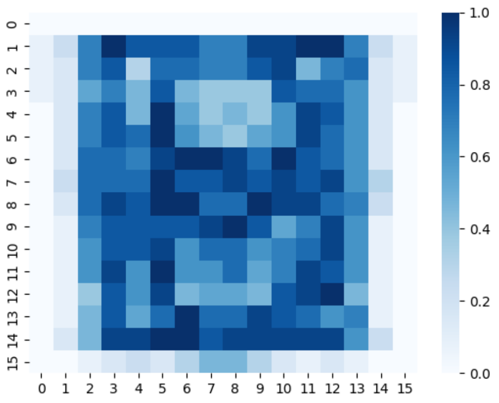
Since I already have all those matrices for every letter, I can add all of the matrices and obtain a frequency map. That frequency map will give me the information for every pixel of the number of letters that have that pixel black and the number of letters that have it white.

Each pixel separates each letter into two sets, black and white. The pixels that give the most information are the ones that divides the images into two sets of equal size (26/2 = 13).

Knowing that, I built a weight map accordingly to reward the pixels that give us more information.



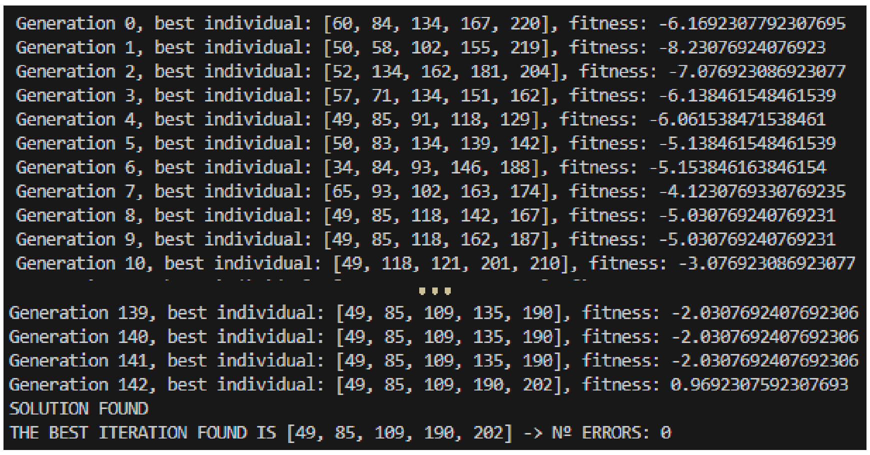
FREQUENCY MAP



WEIGHT MAP

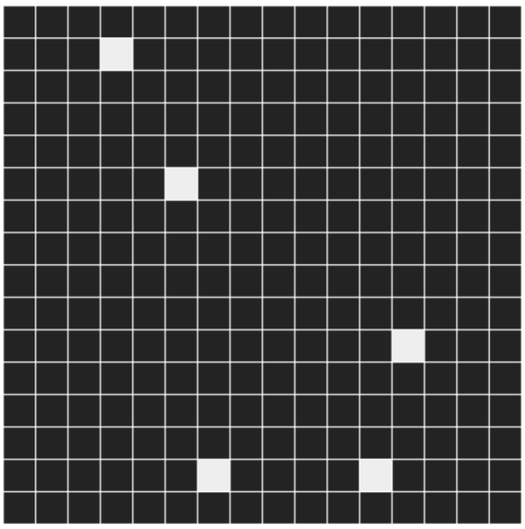
Now I have to add this modification into the fitness function. For every individual, I will add for each pixel its weight map position divided by 5. Now I will slightly larger fitness functions but, because I divided by 5, the number of errors will always be the floor of the fitness.

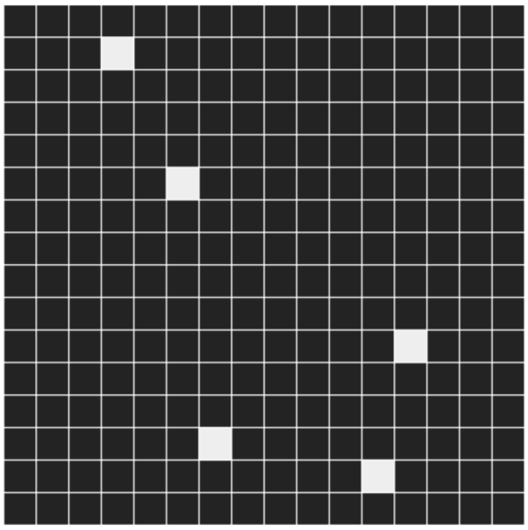
After some iterations now we can expect to get something like this:



```
 Generation 0, best individual: [60, 84, 134, 167, 220], fitness: -6.1692307792307695
 Generation 1, best individual: [50, 58, 102, 155, 219], fitness: -8.23076924076923
 Generation 2, best individual: [52, 134, 162, 181, 204], fitness: -7.076923086923077
 Generation 3, best individual: [57, 71, 134, 151, 162], fitness: -6.138461548461539
 Generation 4, best individual: [49, 85, 91, 118, 129], fitness: -6.061538471538461
 Generation 5, best individual: [50, 83, 134, 139, 142], fitness: -5.138461548461539
 Generation 6, best individual: [34, 84, 93, 146, 188], fitness: -5.153846163846154
 Generation 7, best individual: [65, 93, 102, 163, 174], fitness: -4.123076933076923 5
 Generation 8, best individual: [49, 85, 118, 142, 167], fitness: -5.030769240769231
 Generation 9, best individual: [49, 85, 118, 162, 187], fitness: -5.030769240769231
 Generation 10, best individual: [49, 118, 121, 201, 210], fitness: -3.076923086923077
                                      ...
Generation 139, best individual: [49, 85, 109, 135, 190], fitness: -2.0307692407692306
Generation 140, best individual: [49, 85, 109, 135, 190], fitness: -2.0307692407692306
Generation 141, best individual: [49, 85, 109, 135, 190], fitness: -2.0307692407692306
Generation 142, best individual: [49, 85, 109, 190, 202], fitness: 0.9692307592307693
SOLUTION FOUND
THE BEST ITERATION FOUND IS [49, 85, 109, 190, 202] -> Nº ERRORS: 0
```

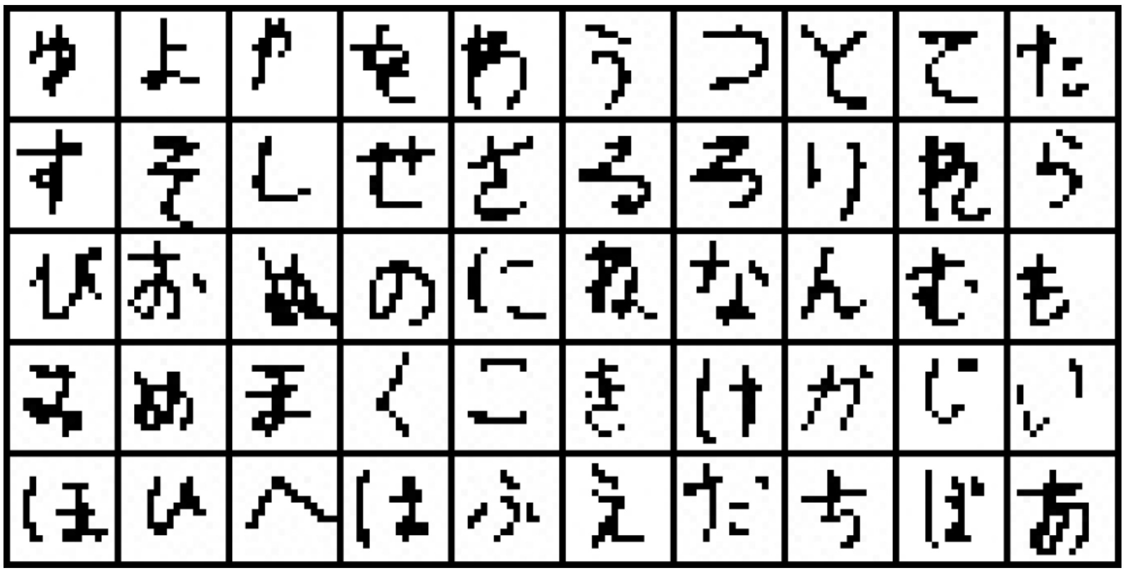
These are the two only solutions I have found:


[49, 85, 110, 190, 202]


[49, 85, 109, 190, 202]

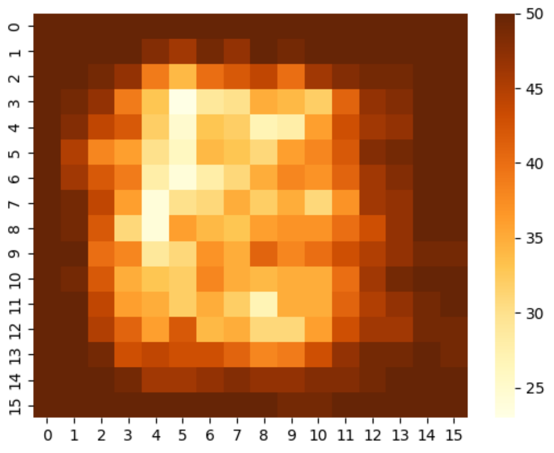
JAPANESE ALPHABET

I will now try to solve the same problem for a Japanese alphabet called hiragana. It contains 50 characters, so we will need at least 6 pixels to solve this problem ($2^5 = 32 < 50 < 64 = 2^6$). The images used are the following:

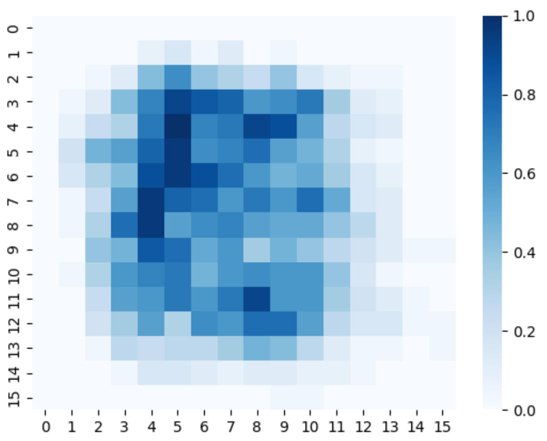


I have already made all the code. Therefore, I only need to make a few corrections to adjust to the new set of images.

The new frequency and weight maps are given by:

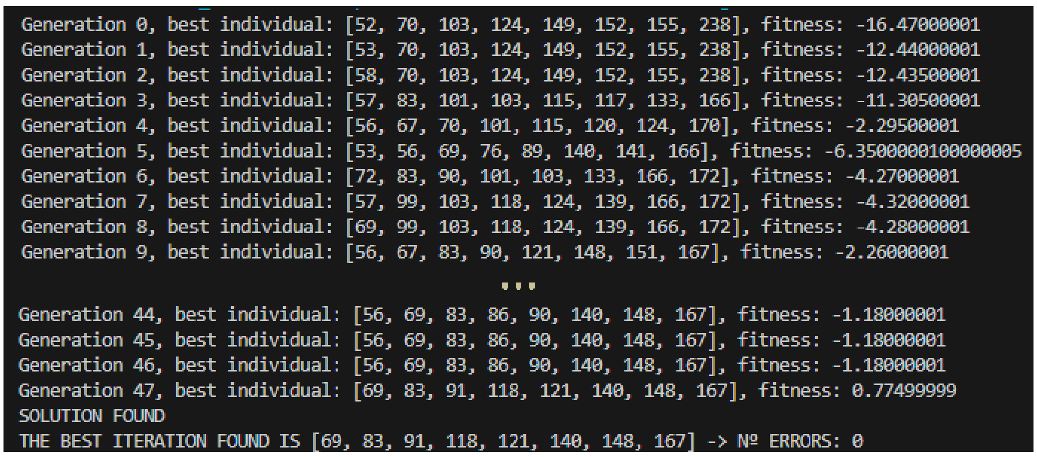


FREQUENCY MAP



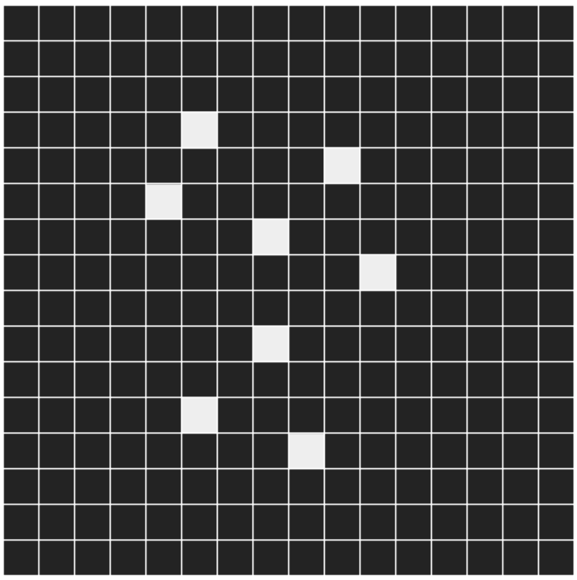
WEIGHT MAP

For 8 pixels, the solution can be found easily:



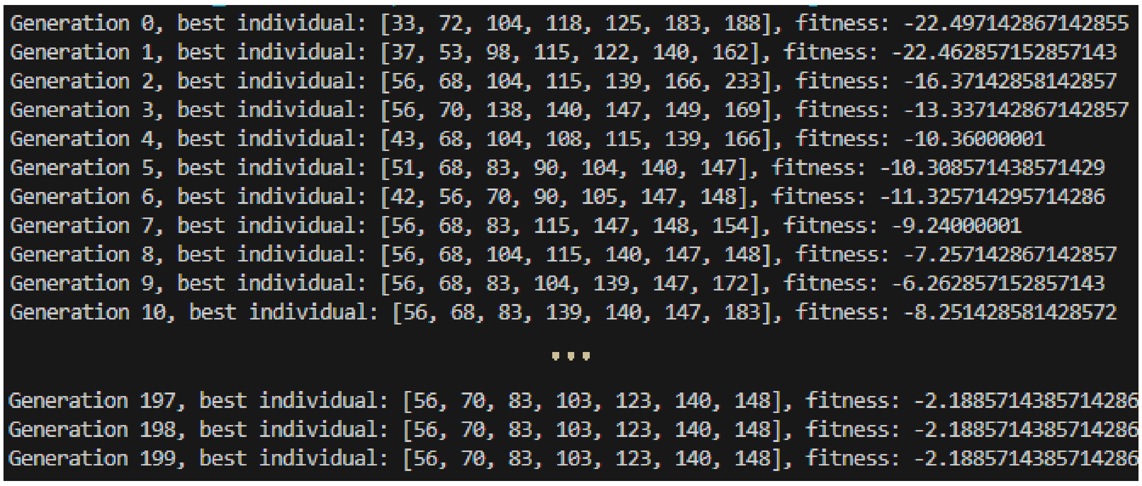
```
Generation 0, best individual: [52, 70, 103, 124, 149, 152, 155, 238], fitness: -16.47000001
Generation 1, best individual: [53, 70, 103, 124, 149, 152, 155, 238], fitness: -12.44000001
Generation 2, best individual: [58, 70, 103, 124, 149, 152, 155, 238], fitness: -12.43500001
Generation 3, best individual: [57, 83, 101, 103, 115, 117, 133, 166], fitness: -11.30500001
Generation 4, best individual: [56, 67, 70, 101, 115, 120, 124, 170], fitness: -2.29500001
Generation 5, best individual: [53, 56, 69, 76, 89, 140, 141, 166], fitness: -6.3500000100000005
Generation 6, best individual: [72, 83, 90, 101, 103, 133, 166, 172], fitness: -4.27000001
Generation 7, best individual: [57, 99, 103, 118, 124, 139, 166, 172], fitness: -4.32000001
Generation 8, best individual: [69, 99, 103, 118, 124, 139, 166, 172], fitness: -4.28000001
Generation 9, best individual: [56, 67, 83, 90, 121, 148, 151, 167], fitness: -2.26000001
                                      ...
Generation 44, best individual: [56, 69, 83, 86, 90, 140, 148, 167], fitness: -1.18000001
Generation 45, best individual: [56, 69, 83, 86, 90, 140, 148, 167], fitness: -1.18000001
Generation 46, best individual: [56, 69, 83, 86, 90, 140, 148, 167], fitness: -1.18000001
Generation 47, best individual: [69, 83, 91, 118, 121, 140, 148, 167], fitness: 0.77499999
SOLUTION FOUND
THE BEST ITERATION FOUND IS [69, 83, 91, 118, 121, 140, 148, 167] -> Nº ERRORS: 0
```

There are many solutions, here I represented one of them:



[69, 83, 91, 118, 121, 140, 148, 167]

However, for 7 pixels the problem is more complex. After many iterations I haven't found any solution, the best solution I have found contains two errors or uncertainties. Here is how a normal iteration looks:



```
Generation 0, best individual: [33, 72, 104, 118, 125, 183, 188], fitness: -22.497142867142855
Generation 1, best individual: [37, 53, 98, 115, 122, 140, 162], fitness: -22.462857152857143
Generation 2, best individual: [56, 68, 104, 115, 139, 166, 233], fitness: -16.37142858142857
Generation 3, best individual: [56, 70, 138, 140, 147, 149, 169], fitness: -13.337142867142857
Generation 4, best individual: [43, 68, 104, 108, 115, 139, 166], fitness: -10.36000001
Generation 5, best individual: [51, 68, 83, 90, 104, 140, 147], fitness: -10.308571438571429
Generation 6, best individual: [42, 56, 70, 90, 105, 147, 148], fitness: -11.325714295714286
Generation 7, best individual: [56, 68, 83, 115, 147, 148, 154], fitness: -9.24000001
Generation 8, best individual: [56, 68, 104, 115, 140, 147, 148], fitness: -7.257142867142857
Generation 9, best individual: [56, 68, 83, 104, 139, 147, 172], fitness: -6.262857152857143
Generation 10, best individual: [56, 68, 83, 139, 140, 147, 183], fitness: -8.251428581428572

                                        ...

Generation 197, best individual: [56, 70, 83, 103, 123, 140, 148], fitness: -2.1885714385714286
Generation 198, best individual: [56, 70, 83, 103, 123, 140, 148], fitness: -2.1885714385714286
Generation 199, best individual: [56, 70, 83, 103, 123, 140, 148], fitness: -2.1885714385714286
```

For 6 pixels the problem is even more complex and even after many iterations, the number of errors or uncertainties is bigger than 10, which is a lot higher that I originally expected.

Even though this algorithm failed heavily to solve the problem for the second set of characters for 6 pixels, it doesn't mean that there is no solution. I tried to prove that there is no solution for this problem using the frequency map computed but I also failed to prove that. There seems to be no way of finding out whether the problem has a solution, since if we wanted to try by brute force we would have to try 256*255*254*253*252*251 different combinations. That is 2.65343618*10^{14}, so it would not be feasible to do. Even though we seem to be pretty far from the solution, taking into account the number of possible combinations there are, I think my algorithm did pretty good.