ΔΕΥΤΕΡΗ ΕΡΓΑΣΙΑ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ
Τμήμα Πληροφορικής

Μάθημα: «ΤΕΧΝΗΤΗ ΝΟΗΜΟΣΥΝΗ ΚΑΙ ΕΜΠΕΙΡΑ ΣΥΣΤΗΜΑΤΑ (6ο εξ.)»

Ομάδα εργασίας:

Π18101 – ΑΝΑΣΤΑΣΙΑ ΙΩΑΝΝΑ ΜΕΞΑ
Π18123 – ΒΑΣΙΛΙΚΗ ΠΑΣΙΑ

Τα γράμματα που σχεδιάζουμε και για τα δύο ερωτήματα της εργασίας είναι Π και Μ. Οι κώδικες υλοποιήθηκαν σε Python.

Άσκηση 1:

➢ Επεξήγηση του γενετικού αλγορίθμου:

Οι γενετικοί αλγόριθμοι αποτελούνται από τα 5 παρακάτω στάδια:

- Δημιουργία τυχαίου αρχικού πληθυσμού
- Συνάρτηση καταλληλότητας
- Επιλογή γονέων
- Αναπαραγωγή
- Μετάλλαξη

Τα παραπάνω βήματα υλοποιούνται στο πρόγραμμά μας με την χρήση συναρτήσεων.

➢ Ανάλυση λειτουργίας συναρτήσεων:

Έχουμε δημιουργήσει συνολικά 9 συναρτήσεις μέσα στον κώδικά μας. Συγκεκριμένα τις:

1. create_population()
2. calculate_fitness()
3. select_best()
4. select_worst()
5. single_point_crossover()
6. mutation()
7. adjust_population()
8. genetic_algo()
9. lists_to_strings()

Παρακάτω αναλύονται οι συναρτήσεις και οι λειτουργίες τους.

1. Η create_population() δημιουργεί τυχαίο πληθυσμό χρωμοσωμάτων μεγέθους size και αριθμό γονιδίων genes_num, όπου size και genes_num είναι τα ορίσματα που δέχεται η συνάρτηση.

```
1 import random
2 from tabulate import tabulate
3
4 # Create random chromosomes
5 def create_population(size, genes_num):
6         population = []
7         for i in range(0,size):
8                 temp = ""
9                 for j in range(0,genes_num):
10                        temp = temp + str(random.getrandbits(1))
11                population.append(temp)
12
13        return population
14
```

2. Η calculate_fitness() υπολογίζει την τιμή της συνάρτησης καταλληλότητας για κάθε χρωμόσωμα (γονέα) του πληθυσμού. Η τιμή αυτή αυξάνεται κάθε φορά που εντοπίζονται ίδια συνεχόμενα bit (γονίδια). Για παράδειγμα στην ακολουθία 110010, η τιμή της συνάρτησης καταλληλότητας είναι ίση με 2, διότι υπάρχει πρόβλημα στα υπογραμμισμένα bit. Επομένως, όσο μεγαλύτερη είναι η

τιμή της συνάρτησης καταλληλότητας, τόσο «ακατάλληλος» είναι ο γονέας και όσο μικρότερη είναι η τιμή, τόσο «κατάλληλος» είναι ο γονέας.

```
15 # Calculates fitness of parents (the population of similar bits in order)
16 def calculate_fitness(population):
17         fitness = []
18         for chromosome in population:
19                 fit_count = 0
20                 for i in range(0, len(chromosome) - 1):
21                         if chromosome[i] == chromosome[i+1]:
22                                 fit_count += 1
23                 print(chromosome,fit_count)
24                 fitness.append(fit_count)
25
26         return fitness
27
```

3. Η select_best() βρίσκει σε ποια θέση του πίνακα (πληθυσμού) υπάρχουν οι 2 καλύτεροι γονείς. Αυτό σημαίνει, ότι εντοπίζει τους γονείς αυτούς με την μικρότερη τιμή στην συνάρτηση καταλληλότητας.

```
28 # Find the index of the best parents (low fitness --> a few of similar bits in order)
29 def select_best(fitness):
30         parent1 = min(fitness)
31         index_parent1 = fitness.index(parent1)
32         fitness[index_parent1] = 10
33         parent2 = min(fitness)
34         index_parent2 = fitness.index(parent2)
35         fitness[index_parent1] = parent1
36
37         return index_parent1, index_parent2
38
```

4. Η select_worst() βρίσκει σε ποια θέση του πίνακα (πληθυσμού) υπάρχουν οι 2 χειρότεροι γονείς. Αυτό σημαίνει, ότι εντοπίζει τους γονείς αυτούς με την μεγαλύτερη τιμή στην συνάρτηση καταλληλότητας.

```
39 # Find the index of the worst parents (high fitness --> a lot of similar bits in order)
40 def select_worst(fitness):
41         parent1 = max(fitness)
42         index_parent1 = fitness.index(parent1)
43         fitness[index_parent1] = -1
44         parent2 = max(fitness)
45         index_parent2 = fitness.index(parent2)
46         fitness[index_parent1] = parent1
47
48         return index_parent1, index_parent2
49
```

5. Η single_point_crossover() παράγει τους απογόνους, με την τεχνική διασταύρωσης ενός σημείου. Δέχεται ως ορίσματα, τον πληθυσμό, την θέση των δύο καλύτερων γονέων του πληθυσμού και το πλήθος των bit που είναι ίσα με 1 στην μάσκα διασταύρωσης (μεταβλητή mask). Επιστρέφει, τους δύο νέους απογόνους που είναι αποτέλεσμα της αναπαραγωγής των γονέων.

```
50 # Implements the single point crossover technique, to create the offsprings
51 def single_point_crossover(population, index_parent1, index_parent2, mask):
52         temp_parent1 = population[index_parent1]
53         temp_parent2 = population[index_parent2]
54         offspring1 = temp_parent1[:mask] + temp_parent2[mask:]
55         offspring2 = temp_parent2[:mask] + temp_parent1[mask:]
56         offspring1 = list(offspring1)
57         offspring2 = list(offspring2)
58 
59         return offspring1, offspring2
60 
```

6. Η mutation() εφαρμόζει μετάλλαξη ενός τυχαία επιλεγόμενου bit και στους δύο απογόνους. Επιστρέφει τους μεταλλαγμένους απογόνους.

```
61 # Applies the mutation of a bit randomly
62 def mutation(offspring1, offspring2):
63         # Choose the random bit
64         mutation_gene_1 = random.randint(0, len(offspring1)-1)
65         mutation_gene_2 = random.randint(0, len(offspring2)-1)
66 
67 
68         if offspring1[mutation_gene_1] == "0":
69                 offspring1[mutation_gene_1] = "1"
70         else:
71                 offspring1[mutation_gene_1] = "0"
72 
73         if offspring2[mutation_gene_2] == "0":
74                 offspring2[mutation_gene_2] = "1"
75         else:
76                 offspring2[mutation_gene_2] = "0"
77 
78         offspring1, offspring2 = lists_to_strings(offspring1, offspring2)
79 
80         return mutation_gene_1, mutation_gene_2, offspring1, offspring2
81 
```

7. Η adjust_population() προσαρμόζει τον πληθυσμό αντικαθιστώντας τους δύο χειρότερους γονείς, με τους καινούργιους απογόνους. Επιστρέφει τον καινούργιο πληθυσμό.

```
82 # Creates the new population, replacing the worst parents with offsprings
83 def adjust_population(population, index_parent1, index_parent2, offspring1, offspring2):
84         offspring1, offspring2 = lists_to_strings(offspring1, offspring2)
85         population[index_parent1] = offspring1
86         population[index_parent2] = offspring2
87 
88         return population
89 
```

8. Η genetic_algo() υλοποιεί την διαδικασία του γενετικού αλγορίθμου, δίνοντας τιμή στις μεταβλητές που χρησιμοποιούνται ως ορίσματα και καλώντας με την κατάλληλη σειρά τις συναρτήσεις που προαναφέρθηκαν. Συγκεκριμένα τρέχει μέχρι να φέρει την επιθυμητή λύση για πληθυσμό μεγέθους 5 και χρωμοσώματα μεγέθους n γονιδίων. Το δεύτερο όρισμα c, υποδηλώνει το πλήθος των bit που είναι ίσα με 1 στην μάσκα διασταύρωσης. Ακόμα, πρέπει να σημειωθεί ότι η πιθανότητα να υπάρξει μετάλλαξη είναι ίση με 20% και γίνεται ανανέωση του πληθυσμού της τάξης του 40%. Επιστρέφει την καλύτερη λύση, δηλαδή το χρωμόσωμα με τιμή συνάρτησης καταλληλότητας ίση με 0 και την καλύτερη γενιά.

```
90 # Implementation of the genetic algorithm
91 def genetic_algo(n,c):
92         i = 1 # generation
93         num_iter = 1000 # max number of iterations
94         fittest_found = False
95         best_solution = ""
96         best_generation = 0
97         mutation_prob = 2 # probability of mutation
98 
99         p = create_population(5,n) # create the initial population
100        f = calculate_fitness(p) # calculate the fitness
101        print("\n")
102 
103        while i < num_iter + 1: # run until max iterations is reached
104                print("GENERATION: ", i)
105                ip1, ip2 = select_best(f) # find the best parents
106                of1, of2 = single_point_crossover(p, ip1, ip2, c) # create offsprings
107                # Check if mutation happens
108                if random.randint(1, 10) <= mutation_prob:
109                        print("Mutation!")
110                        m1, m2, of1, of2 = mutation(of1, of2) # create mutated offsprings
111 
112                ip1, ip2 = select_worst(f) # find the worst parents
113                p = adjust_population(p, ip1, ip2, of1, of2) # create the new population
114                f = calculate_fitness(p) # calculate the new fitness
115                print("\n")
116 
117                # If best solution is found
118                if 0 in f and i > 1:
119                        best_generation = i
120                        fittest_found = True
121                        k = 0
122                        for j in f:
123                                if j == 0:
124                                        best_solution = p[k] # returns the solution
125                                k += 1
126                        break
127 
128 
129                i += 1 # move to the next generation
130 
131        return  fittest_found, best_generation, best_solution
132 
```

9. Τέλος, η lists_to_strings() όπως λέει και το όνομά της, μετατρέπει μια λίστα σε string.

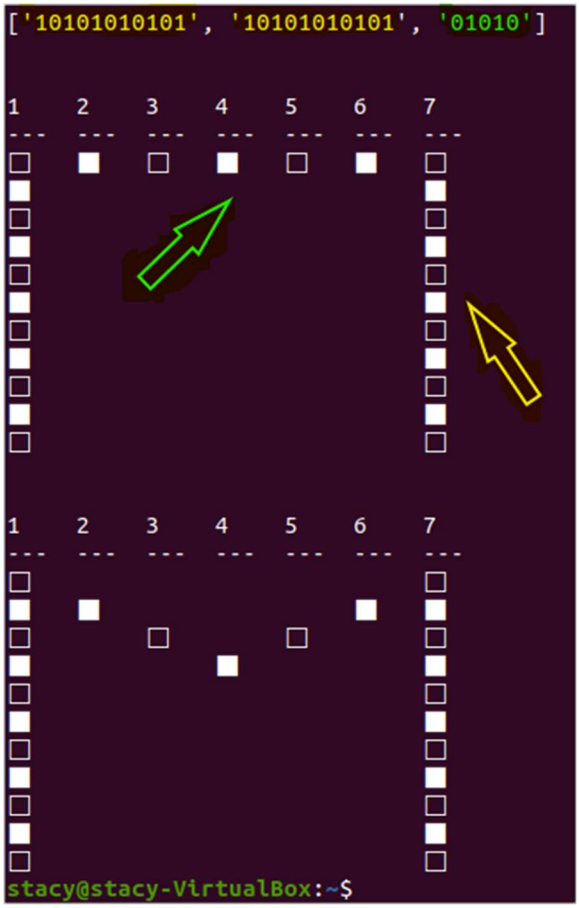
```
133 def lists_to_strings(lst1, lst2): # self explanatory...
134         str1 = ""
135         for c in lst1:
136                 str1 += str(c)
137         str2 = ""
138         for c in lst2:
139                 str2 += str(c)
140 
141         return str1, str2
142 
```

➢ Το κύριο κομμάτι του προγράμματος:

```
# Main
num_of_genes = 11
mask = 5  # crossover mask
solutions = []
num_of_solutions = 0
bit = 0 # the first bit of the solution with 11 number of genes
while True:
        if num_of_solutions == 0:
                # Run the genetic algorithm
                fittest_found, best_generation, best_solution = genetic_algo(num_of_genes,mask)

                # If max number of iterations is reached, run again
                while fittest_found == False:
                        print("Reached max iterations!\n")
                        fittest_found, best_generation, best_solution = genetic_algo(num_of_genes,mask)
        elif num_of_solutions == 2:
                while bit == int(solutions[0][0]):
                        # Run the genetic algorithm
                        fittest_found, best_generation, best_solution = genetic_algo(num_of_genes,mask)
                        bit = int(best_solution[0])
                        # If max number of iterations is reached, run again
                        while fittest_found == False:
                                print("Reached max iterations!\n")
                                fittest_found, best_generation, best_solution = genetic_algo(num_of_genes,mask)
                                bit = int(best_solution[0])
                        if bit != int(solutions[0][0]): # if bits are compatible
                                solutions.append(best_solution)
                                num_of_solutions += 1


        # If optimal solution found
        if fittest_found == True:
                if num_of_solutions == 0:
                        for i in range(0,2):
                                solutions.append(best_solution)
                                num_of_solutions += 1
                        if int(solutions[0][0]) == 1:
                                bit = 1
                print("Optimal solution found on generation: ", best_generation)
                print("Best solution: ", best_solution)
                if num_of_solutions == 2:
                        num_of_genes = 5
                        mask = 3
                if num_of_solutions == 3:
                        break

print(solutions)

# Setting the 11 x 7 grid
table = []
table.append(["","  ","  ","  ","  ","  ","  ",""])
table.append(["","  ","  ","  ","  ","  ","  ",""])
table.append(["","  ","  ","  ","  ","  ","  ",""])
table.append(["","  ","  ","  ","  ","  ","  ",""])
table.append(["","  ","  ","  ","  ","  ","  ",""])
table.append(["","  ","  ","  ","  ","  ","  ",""])
table.append(["","  ","  ","  ","  ","  ","  ",""])
table.append(["","  ","  ","  ","  ","  ","  ",""])
table.append(["","  ","  ","  ","  ","  ","  ",""])
table.append(["","  ","  ","  ","  ","  ","  ",""])
table.append(["","  ","  ","  ","  ","  ","  ",""])


# Replacing ones and zeros
k = 0
for i in solutions:
        str = ""
        for j in solutions[k]:
                if j == "0":
                        str = str + "■"
                elif j == "1":
                        str = str + "□"
        solutions[k] = str
        k += 1
print("\n")
```

```
# Same for P and M
j = 0
for i in solutions[0]:
        table[j][0] = i
        table[j][6] = i
        j += 1

# For P
j = 1
for i in solutions[2]:
        table[0][j] = i
        j += 1

print(tabulate(table, headers=['1', '2', '3', '4', '5', '6','7']))
print("\n")

# For M
j = 1
for i in solutions[2]:
        table[0][j] = " "
        j += 1


table[1][1] = solutions[2][0]
table[2][2] = solutions[2][1]
table[3][3] = solutions[2][2]
table[2][4] = solutions[2][3]
table[1][5] = solutions[2][4]

print(tabulate(table, headers=['1', '2', '3', '4', '5', '6','7']))
```

Στην αρχή ορίζουμε το πλήθος των γονιδίων που θα περιέχει το κάθε χρωμόσωμα με 11, το πλήθος των bit που είναι ίσα με 1 στην μάσκα διασταύρωσης με 5 και ξεκινάει η διαδικασία. Θέλουμε να βρούμε μια λύση με τα παραπάνω στοιχεία πληθυσμού που δώσαμε, με σκοπό να σχηματίσουμε τις δύο στήλες των γραμμάτων Μ και Π. Μόλις εξασφαλίσουμε αύτη την λύση, τρέχουμε ξανά τον γενετικό αλγόριθμο, ωστόσο αλλάζοντας το πλήθος των γονιδίων με 5 και την μάσκα με 3, για να βρούμε την λύση που θα μας σχεδιάσει την πάνω γραμμή που ενώνει τις δύο στήλες των γραμμάτων Μ και Π. Στο παρακάτω σχήμα διακρίνονται καλύτερα οι γραμμές που πρέπει να σχηματιστούν.



Με κίτρινο διακρίνεται η πρώτη λύση του γενετικού αλγορίθμου που παράγει τις πλαϊνές στήλες που δείχνει το κίτρινο βελάκι και με πράσινο διακρίνεται η δεύτερη λύση που παράγει τις πάνω γραμμές που δείχνει το πράσινο βελάκι.

Η διαδικασία του αλγορίθμου εκτελείται με max αριθμό επαναλήψεων (γενεών) τις 1000. Αν δεν έχει βρεθεί λύση μέχρι τότε, ο αλγόριθμος εκτελείται από την αρχή. Αν συμβεί κάτι τέτοιο, στον χρήστη εμφανίζεται ως εξής:

```
GENERATION:  1000
01001010101 1
01001010101 1
01001010101 1
01001010101 1
01001010101 1


Reached max iterations!

01101110001 5
11000101011 4
00011100010 6
00001111001 7
00110110010 4
```

Επίσης, διασφαλίζουμε πάντα πως το πρώτο bit της δεύτερης λύσης θα είναι πάντα διαφορετικό από αυτό της πρώτης λύσης, διότι αλλιώς θα χαλάσει η αντιστοιχία των χρωμάτων στο αποτέλεσμα. Αν η δεύτερη λύση που θα επιστραφεί έχει ίδιο bit με αυτό της πρώτης λύσης, τότε ο αλγόριθμος ξανατρέχει μέχρι να επιστραφεί διαφορετικό bit.

```
['01010101010', '01010101010', '10101']
```

```
['10101010101', '10101010101', '01010']
```

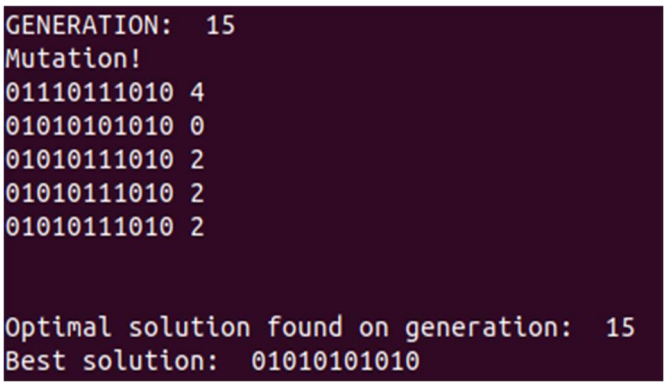
- Επεξήγηση των αποτελεσμάτων:
  Στον χρήστη εκτυπώνονται σε πραγματικό χρόνο ο πληθυσμός κάθε γενιάς, καθώς και αν έχει υπάρξει μετάλλαξη. Ακολουθούν ενδεικτικά screenshots:

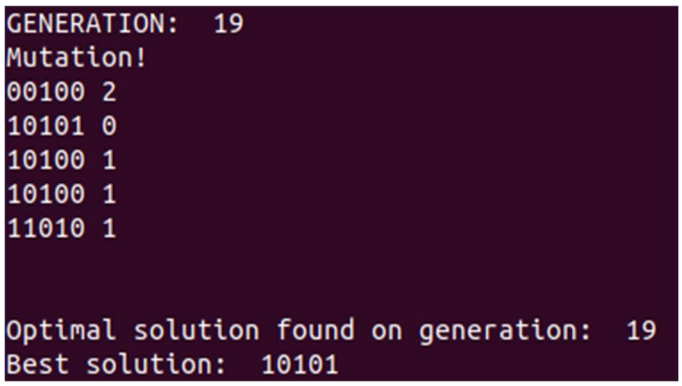
```
11001001101 4
01100111010 4
01111110110 6
01100111111 7
00110011101 5


GENERATION:  1
11001001101 4
01100111010 4
01100001101 5
11001111010 5
00110011101 5


GENERATION:  2
11001001101 4
01100111010 4
11001111010 5
01100001101 5
00110011101 5
```

```
GENERATION:  13
Mutation!
11001001101 4
01100111010 4
11011111010 5
00100001101 5
00110011101 5


GENERATION:  14
11001001101 4
01100111010 4
11001111010 5
01100001101 5
00110011101 5


GENERATION:  15
Mutation!
11001001101 4
01100111010 4
11001011010 3
01100101101 3
00110011101 5
```

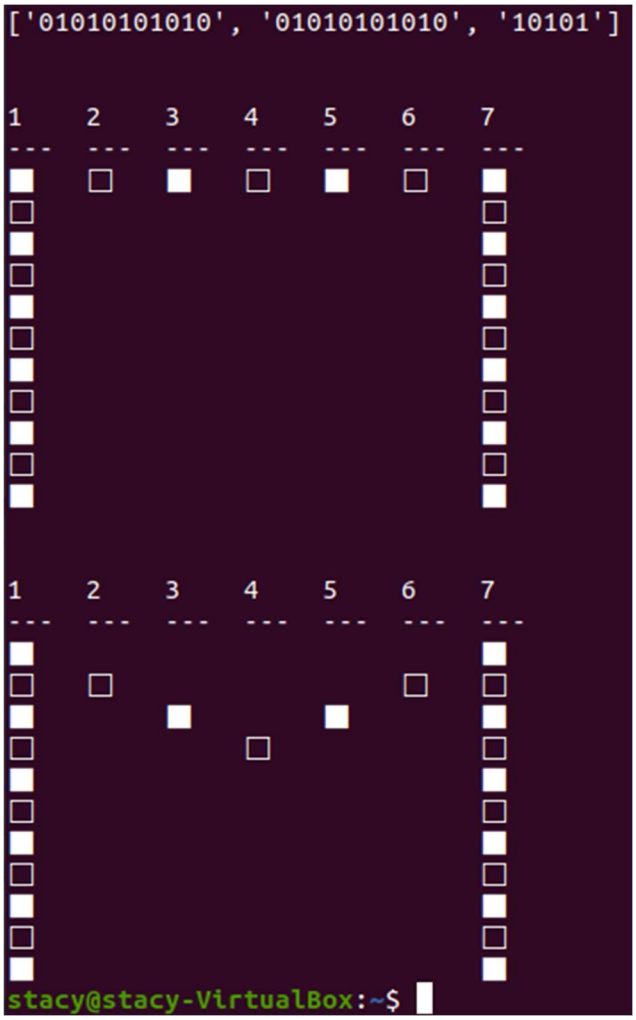
```
10110 1
00000 4
10100 1
10100 1
11010 1


GENERATION:  1
Mutation!
11110 3
10110 1
10100 1
10100 1
11010 1


GENERATION:  2
10100 1
10110 1
10100 1
10100 1
11010 1
```

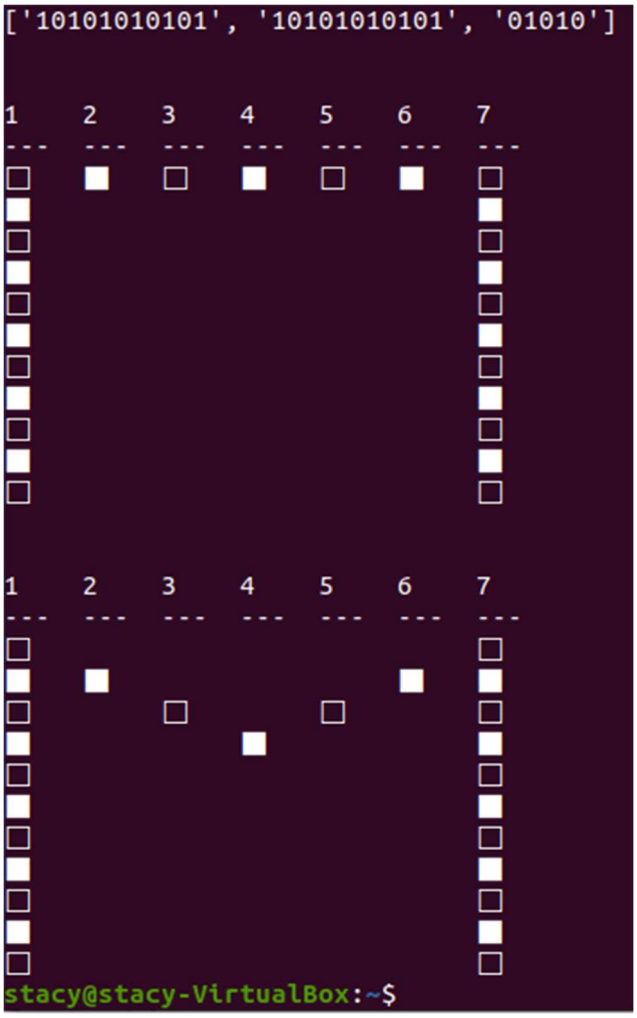
Επιπλέον, ο χρήστης ενημερώνεται όταν βρεθεί η λύση, ποια είναι αυτή και σε ποια γενιά βρέθηκε:





Τέλος, τυπώνεται η τελική λίστα με τις λύσεις και σχεδιάζονται καταλλήλως τα γράμματα Π και Μ:





Άσκηση 2:

- Δημιουργία dataset:
  Έχουμε σχεδιάσει εμείς συνολικά 5 εικόνες μεγέθους 300x300 pixels για κάθε γράμμα και τα χρησιμοποιούμε ως δεδομένα εκπαίδευσης του νευρωνικού δικτύου και μια έκτη για κάθε γράμμα για να τεστάρουμε το δίκτυο.

- Ανάλυση λειτουργίας συναρτήσεων:

Έχουμε δημιουργήσει συνολικά 3 συναρτήσεις μέσα στον κώδικά μας. Συγκεκριμένα τις:

1. sigmoid()
2. sigmoid_derivative()
3. convertImg()

Παρακάτω αναλύονται οι συναρτήσεις και οι λειτουργίες τους.

1. Η sigmoid() στην ουσία υπολογίζει την τιμή της συνάρτησης ενεργοποίησης, ο τύπος της είναι:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

```
1 from PIL import Image, ImageOps
2 import numpy as np
3
4
5 # Activation func
6 def sigmoid(x):
7     return 1 / (1 + np.exp(-x))
8
9
```

2. Η sigmoid_derivative() αντίστοιχα, υπολογίζει την τιμή της παραγώγου της συνάρτησης ενεργοποίησης σύμφωνα με τον κανόνα της αλυσίδας, ο τύπος της είναι:

$$\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$$

```
10 def sigmoid_derivative(x):
11     return x * (1 - x)
12
13
```

3. Η convertImg() υλοποιεί την διαδικασία ψηφιοποίησης της εικόνας σε binary array.

```
14 # convert image to binary array
15 def convertImg(img):
16     grayscale_img = img.convert('L')
17     grayscale_img = grayscale_img.resize((10, 10))  # gives 10x10 pixels img
18     arr = np.array(grayscale_img)  # gives 10x10 array
19     arr = np.reshape(arr, (100, 1))  # gives 100x1 array
20     threshold = 250  # value for determining whether 1 or 0
21     bin_arr = arr
22     for i in range(100):
23         if arr[i][0] > 250:
24             bin_arr[i][0] = 1 # white
25         else:
26             bin_arr[i][0] = 0 # black
27     return bin_arr
28
29
```

- Το κύριο κομμάτι του προγράμματος:

```
30 letter1_bin_arr = list()  # stores input arrays of first letter
31 for i in range(1, 6):
32     letter1_bin_arr.append(convertImg(Image.open('p' + str(i) + '.png')))
33
34 letter2_bin_arr = list()  # stores input arrays of second letter
35 for i in range(1, 6):
36     letter2_bin_arr.append(convertImg(Image.open('m' + str(i) + '.png')))
37
38 training_outputs1 = np.array([[1, 0]]).T  # correct o/p for letter1
39 training_outputs2 = np.array([[0, 1]]).T  # correct o/p for letter2
40
41 learning_rate = 0.01
42
43 synaptic_weights1 = 2*np.random.random(
44     (16, 100)) - 1  # gives a 16x100 array for weights btw first hidden layer and input layer
45 synaptic_weights2 = 2*np.random.random(
46     (16, 16)) - 1  # gives a 16x16 array for weights btw first hidden layer and input layer
47 synaptic_weights3 = 2*np.random.random(
48     (2, 16)) - 1 # gives a 2x16 array for weights btw first hidden layer and output layer
49
```

```
50 acceptable_error1 = False
51 acceptable_error2 = False
52
53 epochs = 0
54
```

Αρχικά, ψηφιοποιούμε όλες τις εικόνες του dataset και φτιάχνουμε τα training set για τα 2 γράμματα. Δημιουργούμε τα πρώτα τυχαία βάρη και προετοιμαζόμαστε για να τρέξει η επανάληψη, ώστε να εκπαιδευτεί το νευρωνικό δίκτυο. Σε κάθε επανάληψη (epoch), υπολογίζουμε τα καινούργια βάρη και το error, αφαιρώντας από την πραγματική έξοδο την επιθυμητή έξοδο και για τα 2 γράμματα. Επομένως, χρησιμοποιούμε back propagation για την αναπροσαρμογή των βαρών και ελαχιστοποιούμε το μέσο τετραγωνικό σφάλμα. Μόλις το συνολικό error και για τα 2 γράμματα γίνει μικρότερο από 0.0001, τερματίζουμε την διαδικασία εκπαίδευσης και προχωράμε στην διαδικασία του testing.

```
while True:

    for bin_arr in letter1_bin_arr:
        hidden_output1 = sigmoid(np.dot(synaptic_weights1, bin_arr))  # gives w(i)*x(i) = (16x100)*(100x1) = (16x1)
        hidden_output2 = sigmoid(np.dot(synaptic_weights2, hidden_output1))  # gives w'(i)*z1(i) = (16x16)*(16x1) = (16x1)
        output = sigmoid(np.dot(synaptic_weights3, hidden_output2))  # gives w''(i)*z2(i) = (2x16)*(16x1) = (2x1)

        error = training_outputs1 - output
        if (error[0] ** 2)/2 + (error[1] ** 2)/2  < 0.0001:
            acceptable_error1 = True
            break

        # Computes the new weights
        dely = (training_outputs1 - output) * sigmoid_derivative(output)
        delz = np.dot(synaptic_weights3.T, dely) * sigmoid_derivative(hidden_output2)
        delx = np.dot(synaptic_weights2.T , delz) * sigmoid_derivative(hidden_output1)
        synaptic_weights3 += (learning_rate * np.dot(dely,hidden_output2.T))
        synaptic_weights2 += (learning_rate * np.dot(delz, hidden_output1.T))
        synaptic_weights1 += (learning_rate * np.dot(delx, bin_arr.T))

    for bin_arr in letter2_bin_arr:
        hidden_output1 = sigmoid(np.dot(synaptic_weights1, bin_arr))  # gives w(i)*x(i) = (16x100)*(100x1) = (16x1)
        hidden_output2 = sigmoid(np.dot(synaptic_weights2, hidden_output1))  # gives w'(i)*z1(i) = (16x16)*(16x1) = (16x1)
        output = sigmoid(np.dot(synaptic_weights3, hidden_output2))  # gives w''(i)*z2(i) = (2x16)*(16x1) = (3x1)

        error = training_outputs2 - output
        if (error[0] ** 2)/2 + (error[1] ** 2)/2 < 0.0001:
            acceptable_error2 = True
            break


         dely = (training_outputs2 - output) * sigmoid_derivative(output)
         delz = np.dot(synaptic_weights3.T, dely) * sigmoid_derivative(hidden_output2)
         delx = np.dot(synaptic_weights2.T , delz) * sigmoid_derivative(hidden_output1)
         synaptic_weights3 += (learning_rate * np.dot(dely,hidden_output2.T))
         synaptic_weights2 += (learning_rate * np.dot(delz, hidden_output1.T))
         synaptic_weights1 += (learning_rate * np.dot(delx, bin_arr.T))

    epochs += 1

    if acceptable_error1 and acceptable_error2:
        break


# test sample
img1 = Image.open('m6.png')
binary_arr1 = convertImg(img1)
input_layer1 = binary_arr1
outputs1 = sigmoid(np.dot( synaptic_weights1 , input_layer1 ))
input_layer2 = outputs1
outputs2 = sigmoid(np.dot( synaptic_weights2 , input_layer2 ))
input_layer3=outputs2
outputs3 = sigmoid(np.dot( synaptic_weights3 , input_layer3 ))

print("Possibility of being a Π:",outputs3[0])
print("Possibility of being a M:",outputs3[1],"\n")
if list(outputs3).index(max(outputs3)) == 0:
    print("It's a Π")
else:
    print("It's a M")
```

Στην διαδικασία testing, φορτώνουμε μια εικόνα Π ή Μ, την μετατρέπουμε σε binary array και υπολογίζουμε τα βάρη. Τέλος, εκτυπώνουμε ποιες είναι οι πιθανότητες να είναι το ένα γράμμα ή το άλλο και τέλος τυπώνεται το γράμμα που προέβλεψε το νευρωνικό δίκτυο.

- Παραδείγματα εκτύπωσης και για τα 2 γράμματα:

```
stacy@stacy-VirtualBox:~/letters$ python3 letters.py
Possibility of being a Π: [0.98061898]
Possibility of being a M: [0.02440245]

It's a Π
```

```
stacy@stacy-VirtualBox:~/letters$ python3 letters.py
Possibility of being a Π: [0.03468627]
Possibility of being a M: [0.96906874]

It's a M
```