ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΚΡΟΙΤΟΡ ΚΑΤΑΡΤΖΙΟΥ ΙΩΑΝ Π21077

ΠΡΟΑΙΡΕΤΙΚΗ ΕΡΓΑΣΙΑ ΜΑΘΗΜΑΤΟΣ ΤΕΧΝΗΤΗΣ ΝΟΗΜΟΣΥΝΗ ΚΑΙ ΕΜΠΕΙΡΑ ΣΥΣΤΗΜΑΤΑ

ΠΕΙΡΑΙΑΣ

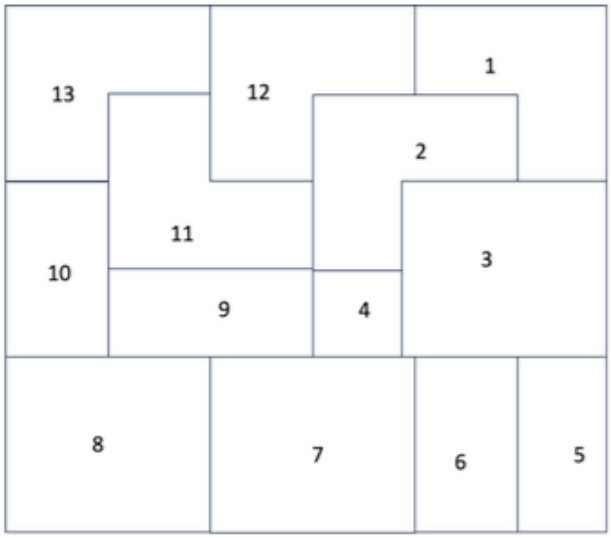
Ιούνιος 2024

ΠΡΟΛΟΓΟΣ – ΕΙΔΙΚΟ ΘΕΜΑ

Η παρούσα εργασία αναπτύχθηκε ως μέρος του μαθήματος Τεχνητής Νοημοσύνη καιΈμπειρα Συστήματα, με κύριο στόχο τη δημιουργία ενός γενετικού αλγόριθμου για την επίλυση μίας ειδικής περίπτωσης του προβλήματος χρωματισμού γράφων.

ΕΚΦΩΝΗΣΗ

Α. Για φοιτητές με επώνυμο από Α-Κ. Αναπτύξτε πρόγραμμα χρωματισμού του παρακάτω γράφου με χρήση γενετικών αλγορίθμων και γλώσσα προγραμματισμού της επιλογής σας. Τα διαθέσιμα χρώματα είναι 4: μπλε, κόκκινο, πράσινο, κίτρινο.



Χρησιμοποιείστε τυχαίο αρχικό πληθυσμό με πλήθος της δικής σας επιλογής. Χρησιμοποιείστε συνάρτηση καταλληλόλητας και διαδικασία επιλογής γονέων σας της δικής σας επιλογής, επίσης. Χρησιμοποιείστε αναπαραγωγή με διασταύρωση ενός σημείου. Επιλέξτε αν θέλετε να κάνετε και μερική ανανέωση πληθυσμού σε κάποιο ποσοστό π.χ. 30% και μετάλλαξη ενός ψηφίου π.χ. στο 10% του πληθυσμού.

ΠΕΡΙΕΧΟΜΕΝΑ

1. ΕΙΣΑΓΩΓΗ

1.1 ΣΤΟΧΟΙ ΕΡΓΑΣΙΑΣ

Βασικός στόχος της εργασίας είναι η υλοποίηση ενός γενετικού αλγορίθμου ο οποίος να χρωματίζει κατάλληλα τον παραπάνω γράφο, ικανοποιώντας παράλληλα τις συνθήκες που αναφέρονται (π.χ. χρησιμοποιώντας 4 χρώματα).

2. ΣΥΝΤΟΜΗ ΠΑΡΟΥΣΙΑΣΗ ΤΟΥ ΠΡΟΒΛΗΜΑΤΟΣ

Στη Θεωρία γράφων ο χρωματισμός ακμών ενός γραφήματος είναι η τοποθέτηση "χρωμάτων" στα άκρα του γραφήματος, έτσι ώστε να μην υπάρχουν δύο γειτονικές ακμές με το ίδιο χρώμα. Ο χρωματισμός ακμών είναι ένας από τους ποικίλους τρόπους χρωματισμού γραφημάτων. Το πρόβλημα χρωματισμού ακμών διερευνά αν είναι εφικτό να χρωματίσουμε τις ακμές ενός δοθέντος γραφήματος χρησιμοποιώντας μέχρι k διαφορετικά χρώματα, όπου k μια δεδομένη τιμή, ή με όσο το δυνατόν λιγότερα χρώματα. (Wikipedia)

3. ΠΕΡΙΓΡΑΦΗ ΠΡΟΓΡΑΜΜΑΤΟΣ

3.1 ΠΑΡΟΥΣΙΑΣΗ ΑΡΧΙΚΗΣ ΣΚΕΨΗΣ

Η βασική απαίτηση του προβλήματος αφορά το γεγονός ότι χρησιμοποιώντας k=4 χρώματα θα πρέπει να χρωματιστεί ο κάθε ένας από τους 13 κόμβους του προκείμενου γράφου, με τέτοιον τρόπο ώστε να μην υπάρχουν δύο γειτονικοί κόμβοι i, j χρωματισμένοι με ίδιο χρώμα. Δοσμένου του γράφου, λοιπόν θα πρέπει να αναπαρασταθεί με κάποιο τρόπο ώστε να περαστεί ως είσοδος στον αλγόριθμο, και αυτός είναι η μήτρα γειτνίασής του (adjacency_matrix). Έτσι, προκύπτει η παρακάτω μήτρα γειτνίασης A(13x13):

0 1 1 0 0 0 0 0 0 0 0 1 0
1 0 1 1 0 0 0 0 0 0 1 1 0
1 1 0 1 1 1 1 0 0 0 0 0 0
0 1 1 0 0 0 1 0 1 0 0 0 0
0 0 1 0 0 1 0 0 0 0 0 0 0
0 0 1 0 1 0 1 0 0 0 0 0 0
0 0 1 1 0 1 0 1 1 0 0 0 0
0 0 0 0 0 0 1 0 1 1 0 0 0
0 0 0 1 0 0 1 1 0 1 1 0 0
0 0 0 0 0 0 0 1 1 0 1 0 1
0 1 0 1 0 0 0 0 1 1 0 1 1
1 1 0 0 0 0 0 0 0 0 1 0 1
0 0 0 0 0 0 0 0 0 1 1 1 0

Η μήτρα είναι μη μηδενική στην θέση A[i,j] εάν ο κόμβος i γειτονεύει με τον κόμβο j. Αυτή αποθηκεύτηκε σε ένα .txt αρχείο και θα φορτωθεί, στην αρχή του προγράμματος.

Αφού αναπαραστάθηκε κατάλληλα ο γράφος, θα πρέπει να δημιουργηθεί ο αλγόριθμος, ο οποίος θα έχει τον ακόλουθο ψευδοκώδικα.

```
begin
    generation = 0
    while ( best_fitness != 0 ):
        selection(population)
        crossover(population)
        mutation(population)
        if ( Best(population) < best_fitness ):
            then best_fitness = Best(population)
        generation += 1
    end while
    return best_fitness
end
```

Σε κάθε νέα γενιά θα πρέπει να επιλεχθούν οι καταλληλότεροι υποψήφιοι (γονείς) από τον αρχικό πληθυσμό (population) σύμφωνα με την συνάρτηση selection(). Ύστερα, από τους γονείς θα προκύψουν νέα παιδιά με την συνάρτηση crossover(), τα οποία στη συνέχεια (πιθανώς) θα μεταλλαχθούν μέσω της συνάρτησης crossover(). Τέλος, ενημερώνεται η τιμή του καλύτερου fitness_score εάν είναι καλύτερη από αυτήν της προηγούμενης γενιάς. Στόχος είναι η ελαχιστοποιήση (minimization) της τιμής fitness_score.

3.2 ΑΝΑΛΥΣΗ ΠΡΟΓΡΑΜΜΑΤΟΣ

Για την ανάπτυξη του γενετικού αλγορίθμου, χρησιμοποιήθηκε η Pyhton3. Αρχικά θα δοθούν κάποιες βασικές πληροφορίες και στη συνέχεια θα αναλυθούν η βασικές συναρτήσεις και ύστερα πως λειτουργεί σε συνδυασμό με την εξέλιξη των γενεών.

3.2.1 ΓΕΝΙΚΗ ΠΕΡΙΓΡΑΦΗ

Το πρόγραμμα αναπτύχθηκε ώστε να υποστηρίζει παραμετροποίηση για αυτό πολλές παράμετροι λαμβάνονται ως είσοδος στην αρχή του προγράμματος. Συγκεκριμένα,

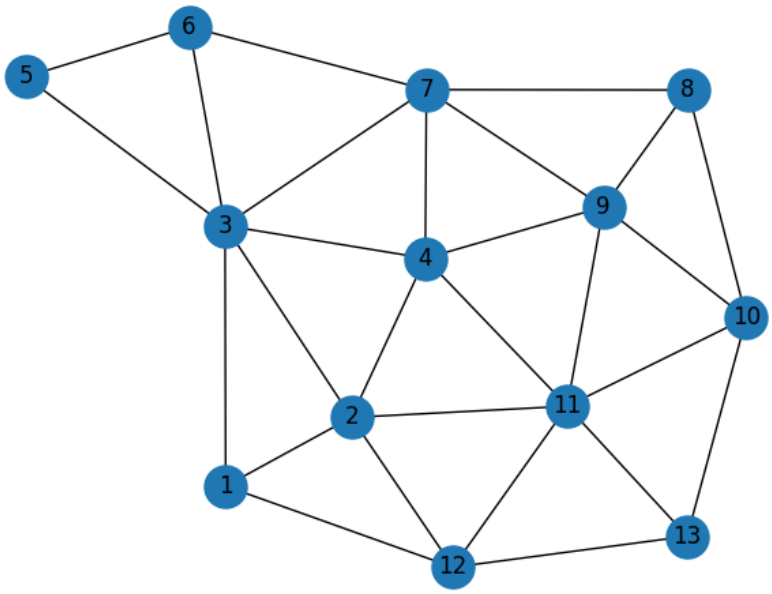
a) Πλήθος χρωμάτων με τα οποία θα χρωματιστούν οι κόμβοι
b) Πλήθος αρχικού πληθυσμού
c) Πιθανότητα μετάλλαξης χρωμοσώματος/ων σε κάθε γενιά
d) Αριθμός χρωμοσωμάτων που θα μεταλλάσσονται σε κάθε γενιά
e) Μέγιστος αριθμός γενιών
f) Αρχικό πλήθος που θα συγκρατηθεί από τον αρχικό πληθυσμό μετά το πέρας της πρώτης γενναίας (μειώνεται κατά 10% σε κάθε γενιά).

Σημειώνεται πως κατά την είσοδο των παραμέτρων από τον χρήστη πραγματεύονται έλεγχοι ασφαλείας, ώστε οι τιμές που εισάγονται να είναι αποδεκτές. Επιπλέον, έλεγχοι πραγματοποιούνται και εντός των συναρτήσεων, για διασφάλιση ομαλής λειτουργίας τους.

Ακριβώς πριν την λήψη των παραμέτρων φορτώνεται η μήτρα γειτνίασης, από την οποία σχεδιάζεται ο γράφος με την matplotlib, όπως φαίνεται παρακάτω.

```
np_matrix = np.array(adjacency_matrix)
print(np_matrix)
rows, cols = np.where(np_matrix == 1)
edges = zip(rows.tolist(), cols.tolist())
G = nx.Graph()
G.add_edges_from(edges)
mylabels = {node: f'{node + 1}' for node in G.nodes()}
nx.draw(G, node_size=500, labels=mylabels ,with_labels=True)
plt.show()
```

Χρησιμοποιούμε την np.where για να βρούμε τις γραμμές και τις στήλες που έχουν τιμή 1, δηλαδή εκεί που υπάρχουν ακμές στον πίνακα γειτνίασης. Μετά, μετατρέπουμε τις γραμμές και τις στήλες σε λίστες και τις συνδυάζουμε σε ζεύγη χρησιμοποιώντας την zip για να δημιουργήσουμε τις ακμές του γραφήματος.



Εικόνα 1. Γράφος προβλήματος

3.2.2 ΑΝΑΛΥΣΗ ΒΑΣΙΚΩΝ ΣΥΝΑΡΤΗΣΕΩΝ

3.2.2.1. generate_population

```
4    # generate population
```

```
5    def generate_population(size: int, power: int, dimensions: int): # size for the # of the generated
     chromosomes
6
7        population = []                                  # power for the # of bits needed to represent the colors
8        for i in range(size):                             # dimensions for the dimensions of the adjacency matrix (#
     of graph nodes)
9            binary_number = []
10           for j in range(power*dimensions):
11               binary_number.append(str(random.randint(0,1)))
12           population.append(''.join(binary_number))
13       return population
14   # blue: 00, red: 01, green: 10, yellow: 11
```

Η συνάρτηση generate_population χρησιμοποιείται για τη δημιουργία ενός αρχικού πληθυσμού χρωμοσωμάτων. Κάθε χρωμόσωμα αναπαρίσταται ως μια συμβολοσειρά δυαδικών ψηφίων (0 και 1).

Παράμετροι:

- *size*: Ο αριθμός των χρωμοσωμάτων που θέλουμε να δημιουργήσουμε.
- *power*: Ο αριθμός των δυαδικών ψηφίων που απαιτούνται για την αναπαράσταση των χρωμάτων. Για παράδειγμα, αν έχουμε τέσσερα χρώματα (μπλε, κόκκινο, πράσινο, κίτρινο), χρειαζόμαστε 2 δυαδικά ψηφία για την αναπαράστασή τους: 00 (μπλε), 01 (κόκκινο), 10 (πράσινο), 11 (κίτρινο).
- *dimensions*: Οι διαστάσεις του πίνακα γειτνίασης, δηλαδή ο αριθμός των κόμβων του γράφου.

3.2.2.2. fitness

```
def fitness(power: int, population: list, adjacency_matrix: list):


    fitness_score_array = []
    splitted_chromosomes = split_chromosomes(power=power, population=population)
    print(splitted_chromosomes)
    colors_matrix = assign_colors(splitted_chromosomes=splitted_chromosomes)
for i in range(len(colors_matrix)):
        fitness_score = 0


        for j in range(len(adjacency_matrix)):
            for k in range(j+1, (len(adjacency_matrix))):
                # check if nodes are adjacent
                if adjacency_matrix[j,k] == 1 and colors_matrix[i][j] == colors_matrix[i][k]: # if the same then increment
the score
                    fitness_score += 1
```

```
            else: # keep the score at it was
                fitness_score = fitness_score
        fitness_score_array.append(fitness_score)

    return fitness_score_array, colors_matrix
```

Η συνάρτηση fitness χρησιμοποιείται για τον υπολογισμό του βαθμού καταλληλότητας (fitness score) κάθε χρωμοσώματος σε έναν πληθυσμό, αξιολογώντας πόσο καλή είναι η κάθε λύση, ώστε να ελαχιστοποιηθεί ο αριθμός των γειτονικών κόμβων που έχουν το ίδιο χρώμα. Ως έξοδος, επιστρέφεται ένα array με το fitness_score του πλυθησμού

Παράμετροι:

- *power*: Ο αριθμός των δυαδικών ψηφίων που απαιτούνται για την αναπαράσταση των χρωμάτων.
- *population*: Μια λίστα με τα χρωμοσώματα που θα αξιολογηθούν.
- *adjacency_matrix*: Ο πίνακας γειτνίασης του γράφου που αναπαριστά ποιες κορυφές είναι γειτονικές.

Διαδικασία

Αφού αναθέσουμε τα κατάλληλα χρώματα για κάθε χρωμόσωμα (blue: 00, red: 01, green: 10, yellow: 11 στον πίνακα colors_matrix ο οποίος δημιουργείται με την συνάρτηση assign_colors), εντός βρόχου ελέγχουμε,

Για κάθε χρωμόσωμα:
Αρχικοποιείται ο βαθμός καταλληλότητας (fitness_score) σε 0.
Για κάθε κόμβο j του γράφου:
Για κάθε επόμενο κόμβο k (ξεκινώντας από τον κόμβο j+1):
Αν οι κόμβοι j και k είναι γειτονικοί (δηλαδή η τιμή στον πίνακα γειτνίασης adjacency_matrix[j, k] είναι 1) και έχουν το ίδιο χρώμα (δηλαδή η τιμή στον πίνακα χρωμάτων colors_matrix[i][j] είναι ίση με την colors_matrix[i][k]):
Αυξάνεται το fitness score κατά 1.

Η συνθήκη αυτή βασίστηκε στη παρακάτω σκέψη:

penalty(i, j) = 1 if there is an edge between i and j
penalty(i, j) = 0 Otherwise

Επομένως το penalty μιας υποψήφιας λύσης θα είναι το άθροισμα της penalty(i,j) σε όλος το μήκος των κόμβων του γράφου.

$$fitness = \sum penalty(i, j)$$

3.2.2.3. tournament_selection

```
4.      # select best solutions (parents) from the current generation to descend to the following ones
5.      def tournament_selection(population: list, fitness_score_array: list, k: int): # k refers to the # of values
        compared each time
6.
7.          indices = [i for i in range(len(fitness_score_array))] # will save the indices of the fitness_score_array
8.          best_parents = [] # best parents in the current population
9.
10.         while len(indices) >= k: # continue until there are enough remaining indices
11.             parents_indices = random.sample(indices, k=2)  # select two unique random indices (if choices
        used then the parents indices may be [3,3], allowing repetitions)
12.             parents = [population[i] for i in parents_indices]  # Get corresponding parents
13.
14.             # select fitness scores for the selected parents according to their indices
15.             fitness_scores = [fitness_score_array[i] for i in parents_indices]
16.
17.             # sort parents based on fitness scores (search for min)
18.             sorted_parents = [parent for _, parent in sorted(zip(fitness_scores, parents), key=lambda x: x[0])] #
        key is the first value of the sorted parents [0]
19.
20.             best_parents.append(sorted_parents[0])
21.
22.             # remove the index of the selected parent from the list of remaining indices
23.             for index in parents_indices:
24.                 indices.remove(index)
25.
26.             # remove the fitness scores of the unselected parents
27.             # for index in indices:
28.             #     fitness_score_array.remove(fitness_score_array[index])
29.
30.         return best_parents
```

Η συνάρτηση αυτή επιλέγει σε κάθε γενιά τους καλύτερους υποψήφιους (γονείς) από τον τρέχοντα πληθυσμό (population) σύμφωνα με το χαμηλότερο fitness score (όσο χαμηλότερο τόσο καλύτερο), τα οποία υπολογίστηκαν σύμφωνα με τις προηγούμενες συναρτήσεις. Στην συνέχεια οι επιλεγμένοι γονείς θα χρησιμοποιηθούν για να διασταυρωθούν.

Παράμετροι:

- *population*: Λίστα με τον τρέχοντα πληθυσμό.
- *fitness_score_array*: Λίστα με τις βαθμολογίες καταλληλόλητας.
- *k*: Μέγεθος του τουρνουά. (πόσοι γονείς θα συγκρίνονται κάθε φορά)

Διαδικασία

Αφού αποθηκευτούν οι θέσεις (indexes) του κάθε fitness_score (που έχουν ακριβής αντιστοιχία με τα indexes της λίστας population), επιλέγονται δύο γονείς προς σύγκριση με την sample(), οποία παράγει δύο μοναδικές τιμές που αναπαριστούν το index των επιλεγμένων γονέων. Στη συνέχεια, η συνάρτηση zip(fitness_scores, parents) συνδυάζει δύο λίστες, τη fitness_scores και τη parents, δημιουργώντας μια λίστα από πλειάδες. Κάθε πλειάδα περιέχει ένα στοιχείο από τη λίστα fitness_scores και το αντίστοιχο στοιχείο από τη λίστα parents. Μετά, η συνάρτηση sorted ταξινομεί τη λίστα των πλειάδων που δημιουργήθηκε, βάσει του πρώτου στοιχείου κάθε πλειάδας (που είναι το fitness_score). Η lambda x: x[0] είναι μια ανώνυμη συνάρτηση που επιστρέφει το πρώτο στοιχείο της πλειάδας για χρήση ως κλειδί ταξινόμησης (δηλαδή ταξινομεί με βάση το fitness_score). Τέλος, το parent for _, parent in sorted(...) εξάγει μόνο τους γονείς από τις ταξινομημένες πλειάδες. Το σύμβολο _ χρησιμοποιείται για να αγνοηθεί η βαθμολογία καταλληλόλητας, αφού μας ενδιαφέρουν μόνο οι γονείς. Ύστερα από το πέρας κάθε επανάληψης τα indexes των γονέων που μόλις συγκρίθηκαν, αφαιρούνται από την λίστα των διαθέσιμων indexes.

3.2.2.4. single_point_crossover

```
4.      def single_point_crossover(best_parents: list):
5.          i=0
6.          # check for same length
7.          while i < len(best_parents) - 1:
8.              if len(best_parents[i]) != len(best_parents[i+1]):
9.                  raise ValueError("\nChromosomes must be of the same length")
10.             i+=1
11.
12.         if len(best_parents) < 2:
13.             return best_parents
14.
15.         crossovered_parents = []
16.
17.         for i in range(0, len(best_parents) - 1, 2): # iterate through all unique pairs of 2 parents
18.             parent1 = best_parents[i]
19.             parent2 = best_parents[i+1]
20.             p = random.randint(1, len(parent1)) # generate a random integer within the bits length
21.             crossovered_parents.append(parent1[0:p]+parent2[p:]) # crossover a
22.             crossovered_parents.append(parent2[0:p]+parent1[p:]) # crossover b
23.
24.         return crossovered_parents
```

Αυτή η συνάρτηση υλοποιεί την διασταύρωση ενός σημείου, κατά την οποία ανά δύο οι γονείς από τον καλύτερο πληθυσμό (ο οποίος επιτεύχθηκε με την βοήθεια τον προηγούμενων συναρτήσεων), διασταυρώνονται, δημιουργώντας δύο παιδιά. Το πρώτο αποτελείται από το πρώτα p bits του πρώτου γονέα και τα τελευταία 1-p bits του δεύτερου γονέα. Αντίστοιχα και

για το δεύτερο παιδί από το πρώτα p bits του δεύτερου γονέα και τα τελευταία 1-p bits του πρώτου γονέα. Ο αριθμός p επιλέγεται τυχαία κάθε φορά.

Παράμετροι

- *best_parents*: Λίστα με του καλύτερους γονείς

3.2.2.5. mutate

```
4.      def mutate(chromosomes: list, mutations_number, probability: float):
5.
6.          if mutations_number > len(chromosomes):
7.              mutations_number = len(chromosomes)
8.
9.          indices = [i for i in range(len(chromosomes))] # will save the indices of the chromosomes
10.         # select a random chromosome to mutate from the given list
11.         chromosome_index = random.randint(0, len(chromosomes)-1)
12.         chromosome = chromosomes[chromosome_index]
13.         chromosome = list(chromosomes[chromosome_index])  # convert string to list for mutability
14.
15.         for _ in range(mutations_number):
16.             index = random.randrange(len(chromosome)) # pick a random bit (index) from the given
        chromosome
17.             if index in indices: # check if the index has already been mutated
18.                 # random generates a random float within [0,1]
19.                 # abs(chromosome[index] - 1) flips the bit from 0 to 1 and from 1 to 0
20.                 chromosome[index] = chromosome[index] if random.random() > probability else
        abs(int(chromosome[index]) - 1)
21.                 indices.remove(index)
22.             else:
23.                 break
24.
25.         # convert list back to string
26.         mutated_chromosome = ''.join(map(str, chromosome))
27.         chromosomes[chromosome_index] = mutated_chromosome  # update the original list
28.
29.         return chromosomes
30.
```

Η τρέχουσα συνάρτηση πραγματοποιεί τυχαίες μεταλλάξεις σε ορισμένα από τα παιδιά που προέκυψαν από την διασταύρωση, σύμφωνα με μία πιθανότητα p, αλλάζοντας n bit από 0 σε 1 και αντίστροφα. Τόσο η πιθανότητα p όσο και ο αριθμός n λαμβάνονται ως είσοδος.

Παράμετροι

- *chromosomes*: Λίστα που περιέχει δυαδικές συμβολοσειρές που αναπαριστούν χρωμοσώματα.
- *mutations_number*: Ο αριθμός μεταλλάξεων που θα εισαχθούν.
- *probability*: Η πιθανότητα κάθε δυαδικού ψηφίου να μεταλλαχθεί

Διαδικασία

Επαναλαμβάνει το εξής n φορές:

- Επιλέγει τυχαία έναν δείκτη (index) από τη λίστα δεικτών.
- Ελέγχει εάν ο επιλεγμένος δείκτης έχει ήδη χρησιμοποιηθεί για μετάλλαξη, και αν ναι, τότε αγνοείται και επιλέγεται ένας νέος δείκτης.
- Αν το αποτέλεσμα της τυχαίας επιλογής (random.random()) είναι μεγαλύτερο από την πιθανότητα (probability), τότε το ψηφίο στον επιλεγμένο δείκτη αντιστρέφεται, δηλαδή από 0 γίνεται 1 και αντίστροφα.
- Αφαιρεί τον επιλεγμένο δείκτη από τη λίστα δεικτών, ώστε να μην μεταλλαχθεί ξανά το ίδιο ψηφίο.

3.2.2.6. main

Αρχικά φορτώνεται το αρχείο adjacency_matrix.txt, δημιουργείται ο γράφος και λαμβάνονται οι απαραίτητες είσοδοι, όπως προαναφέρθηκε. Στη συνέχεια, επαναλαμβάνεται ο εξής βρόχος max_generations φορές:

```
    for generation in range(max_generations):

        print('\nCurrent generation: '+str(generation))
        print('\nRemaining population '+str(population))

        # evaluate current population
        fitness_score_array, colors_matrix = fitness(power=bits, population=population,
adjacency_matrix=np_matrix)
        print('\nFitness scores '+str(fitness_score_array))

        # select best parents
        best_parents = tournament_selection(population=population, fitness_score_array=fitness_score_array, k=2)
        print("\nBest Parents "+str(best_parents))

        # crossover
        crossover = single_point_crossover(best_parents=best_parents)
        print('\nCrossovered '+str(crossover))

        # mutate crossovered children
        mutated_chromosomes = mutate(chromosomes=crossover, mutations_number=mutations_number,
```

```
                                probability=probability)
        print(mutated_chromosomes)

        # evaluate mutated individuals
        mutated_fitness_score_array, _ = fitness(power=bits, population=mutated_chromosomes,
adjacency_matrix=np_matrix)
        print('\nMutated fitness scores '+str(mutated_fitness_score_array))

        population = replace_population(population=population, mutated_chromosomes=mutated_chromosomes,
                        elite_size=elite_size, fitness_score_array=fitness_score_array,
                        mutated_fitness_score_array=mutated_fitness_score_array)

        if len(population) < elite_size:
            print('\nPopulation diminished to minimum')
            break

        colors = colors_matrix[0]
        node_colors = [colors[node] for node in G.nodes()]

        # draw the graph with node colors and labels
        nx.draw(G, node_size=500, labels=mylabels, with_labels=True, node_color=node_colors)
        plt.title('Graph with Node Colors, Generation '+str(generation))
        plt.show()

        elite_size = elite_size - (int(0.1 * elite_size))
        if elite_size < 1:
            elite_size = 1

        if fitness_score_array[0] == 0:
            break
        elif generation == (max_generations - 1):
            break

        generation += 1

    print("\nAlgorithm successfully finished on generation "+str(generation)+
          ' with best coloring being the ' + str(colors_matrix[0])+'\n')
    # draw the graph with node colors and labels
    nx.draw(G, node_size=500, labels=mylabels, with_labels=True, node_color=node_colors)
    plt.title('Best Graph Coloring')
    plt.savefig("best_coloring.png")
```

```
    plt.show()


    return fitness_score_array
```

Κάθε επανάληψη περιλαμβάνει τα εξής βήματα:

Εκτίμηση Τιμής Συνάρτησης Καταλληλότητας:

- Το σύστημα εκτιμά την απόδοση της τρέχουσας γενιάς χρησιμοποιώντας την συνάρτηση καταλληλόλητας αναδεικνύοντας πόσο καλά αποδίδει ο πληθυσμός στο πρόβλημα που επιλύεται.

Επιλογή Καλύτερων Γονέων:

- Το σύστημα επιλέγει τους καλύτερους γονείς από την τρέχουσα γενιά χρησιμοποιώντας μια διαδικασία επιλογής τουρνουά. Αυτοί οι γονείς θα χρησιμοποιηθούν για τη δημιουργία νέων ατόμων.

Διασταύρωση Γονέων:

- Οι επιλεγμένοι γονείς υποβάλλονται σε διασταύρωση ενός σημείου, δημιουργώντας νέα παιδιά.

Μετάλλαξη:

- Τα νέα παιδιά που προέκυψαν από τη διασταύρωση υποβάλλονται σε μεταλλάξεις. Αυτή η διαδικασία αντιστρέφει τυχαία κάποια bit των παιδιών με μια συγκεκριμένη πιθανότητα.

Εκτίμηση Τιμής Συνάρτησης Καταλληλόλητας Μεταλλαγμένων Ατόμων:

- Οι μεταλλαγμένοι γονείς αξιολογούνται ξανά για να μετρήσουν την απόδοσή τους μετά τη μετάλλαξη.

Αντικατάσταση Πληθυσμού:

- Ο πληθυσμός των γονέων συνδυάζεται με αυτόν των μεταλλαγμένων παιδιών. Λαμβάνοντας υπόψη το fitness_score, συγκρατούνται οι καλύτεροι υποψήφιοι (οι ελιτ – ‘elitism’) ώστε να περαστούν στην επόμενη γενιά.

Επιπλέον ο αλγόριθμος τερματίζει αν βρεθεί λύση με fitness_score = 0 ή αν συμπληρωθεί ο μέγιστος αριθμός εποχών, αποθηκεύοντας την εικόνα με τον καλύτερο χρωματισμό του γράφο.

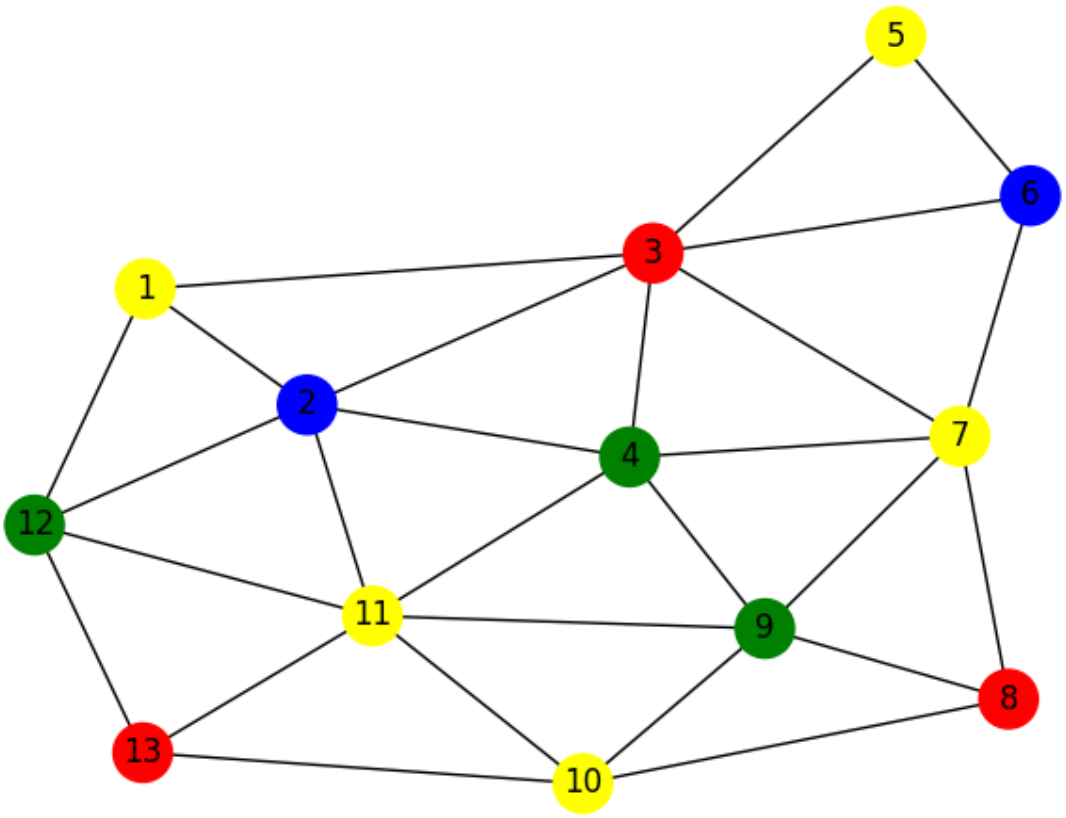
4. ΕΠΙΔΕΙΞΗ ΤΗΣ ΛΥΣΗΣ

Για να αναδείξουμε την λειτουργία του προγράμματος, θα τρέξει ο αλγόριθμος πάνω στο τρέχον πρόβλημα για τις παραμέτρους num_colors=4, population=100, probability=0.3, num_mutations=1, generations=10, elite=50.

Παρακάτω παρατηρούμε τοι output του terminal σε μία τυχαία εποχή generation=2

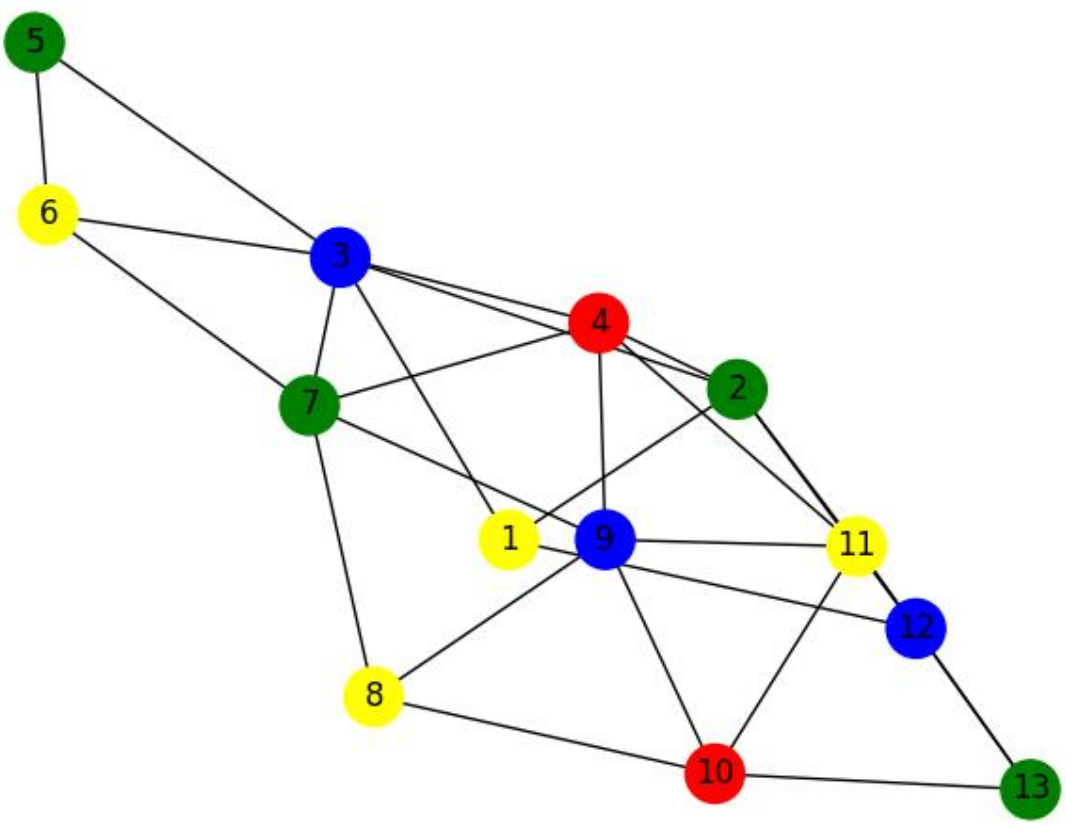
```
Current generation: 2

Remaining population ['1100011011001101101111001', '1110001010111011000111010', '1100011010111011000111010', '1110011011001
0001110010010', '0100101001001100100111110', '0001101111001101101100010', '1001000110111100100111100', '1010110111000111100
1110110', '0111001011111011010010101', '1001000110111100100111100', '1100011011001101110011010', '0100101001001100100111111
0', '0110110111000111100111010', '0010110110011001110111100', '1100011011001101101100100', '0110010011000111100111010', '1
0100001011110111110100100', '0100101001001100100111110', '1000000110111100100111100', '0110011111101000110100111', '0110011
1111011010100100001', '1101001110010000110100111', '0010110110011001110111001', '0101101101000001100100101', '0010110010010
1000111110010', '0010110000101100001100100', '1101011101000010011110100', '0011101001010001101100010', '0110010011000111010
0010010', '1100011000101111100010101', '1000111010101101011010111', '1101001000101011010111011', '0010111010000010111100011
1', '0011100011010011101111001', '1010000101111011110100101', '1010011111101101010010001', '1001000110111100100111001', '1
1010111010000100111101101', '0100101001001100100111011', '1101001110010101010010001', '0001101111001101101010110', '0111001
0111110110101000010', '0110011110100000110101001', '0010000100111000110011001', '1011111100101010010010011']
colorsMatrix [['yellow', 'blue', 'red', 'green', 'yellow', 'blue', 'yellow', 'red', 'green', 'yellow', 'yellow', 'green', 'red
'], ['yellow', 'green', 'blue', 'green', 'green', 'yellow', 'green', 'yellow', 'blue', 'red', 'yellow', 'blue', 'green'], ['ye
llow', 'blue', 'red', 'green', 'green', 'yellow', 'green', 'yellow', 'blue', 'red', 'yellow', 'blue', 'green'], ['yellow', 'gr
een', 'red', 'green', 'yellow', 'blue', 'green', 'blue', 'yellow', 'green', 'red', 'blue', 'green'], ['red', 'blue', 'green',
'green', 'red', 'blue', 'yellow', 'blue', 'green', 'red', 'yellow', 'yellow', 'green'], ['blue', 'red', 'green', 'yellow', 'ye
llow', 'blue', 'yellow', 'red', 'green', 'yellow', 'blue', 'blue', 'green'], ['green', 'red', 'blue', 'red', 'green', 'yellow'
, 'yellow', 'blue', 'green', 'red', 'yellow', 'yellow', 'blue'], ['green', 'green', 'yellow', 'red', 'yellow', 'blue', 'red',
'yellow', 'green', 'red', 'yellow', 'red', 'green'], ['red', 'yellow', 'blue', 'green', 'yellow', 'yellow', 'green', 'yellow',
 'red', 'blue', 'green', 'yellow', 'red'], ['green', 'red', 'blue', 'red', 'green', 'yellow', 'yellow', 'blue', 'green', 'red'
, 'yellow', 'yellow', 'blue'], ['yellow', 'blue', 'red', 'green', 'yellow', 'blue', 'yellow', 'red', 'yellow', 'blue', 'yellow
', 'red', 'red'], ['red', 'blue', 'green', 'green', 'red', 'blue', 'yellow', 'blue', 'green', 'red', 'yellow', 'yellow', 'gree
n'], ['red', 'green', 'yellow', 'red', 'yellow', 'blue', 'red', 'yellow', 'green', 'red', 'yellow', 'red', 'green'], ['blue',
'green', 'yellow', 'red', 'green', 'red', 'green', 'red', 'yellow', 'red', 'yellow', 'yellow', 'blue'], ['yellow', 'blue', 're
d', 'green', 'yellow', 'blue', 'yellow', 'red', 'green', 'yellow', 'blue', 'red', 'blue'], ['red', 'green', 'red', 'blue', 'ye
llow', 'blue', 'red', 'yellow', 'green', 'red', 'yellow', 'red', 'green'], ['green', 'green', 'blue', 'red', 'red', 'yellow',
'green', 'yellow', 'yellow', 'red', 'blue', 'red', 'blue'], ['red', 'blue', 'green', 'green', 'red', 'blue', 'yellow', 'blue',
 'green', 'red', 'yellow', 'yellow', 'green'], ['green', 'blue', 'blue', 'red', 'green', 'yellow', 'yellow', 'blue', 'green',
'red', 'yellow', 'yellow', 'blue'], ['red', 'green', 'red', 'yellow', 'yellow', 'green', 'green', 'blue', 'yellow', 'red', 'bl
ue', 'yellow', 'green'], ['red', 'green', 'red', 'yellow', 'yellow', 'green', 'yellow', 'red', 'red', 'blue', 'green', 'blue',
 'red'], ['yellow', 'red', 'blue', 'yellow', 'green', 'red', 'blue', 'blue', 'yellow', 'red', 'blue', 'yellow', 'green'], ['bl
ue', 'green', 'yellow', 'red', 'green', 'red', 'green', 'red', 'yellow', 'red', 'yellow', 'green', 'red'], ['red', 'red', 'gre
en', 'yellow', 'red', 'blue', 'blue', 'red', 'green', 'red', 'blue', 'green', 'green'], ['blue', 'green', 'yellow', 'blue', 'g
reen', 'red', 'red', 'blue', 'red', 'yellow', 'yellow', 'blue', 'green'], ['blue', 'green', 'yellow', 'blue', 'blue', 'green',
 'yellow', 'blue', 'blue', 'yellow', 'green', 'red', 'blue'], ['yellow', 'red', 'red', 'yellow', 'red', 'blue', 'blue', 'green
', 'red', 'yellow', 'green', 'yellow', 'blue'], ['blue', 'yellow', 'green', 'green', 'red', 'red', 'blue', 'red', 'green', 'ye
llow', 'blue', 'red', 'blue'], ['red', 'green', 'red', 'blue', 'yellow', 'blue', 'red', 'yellow', 'red', 'blue', 'red', 'blue'
, 'green'], ['yellow', 'blue', 'red', 'green', 'blue', 'green', 'yellow', 'yellow', 'green', 'blue', 'green', 'green', 'yellow
'], ['green', 'blue', 'yellow', 'green', 'green', 'green', 'yellow', 'red', 'red', 'green', 'yellow', 'red', 'yellow'], ['yell
ow', 'red', 'blue', 'green', 'blue', 'green', 'green', 'yellow', 'red', 'green', 'yellow', 'green', 'yellow'], ['blue', 'green
', 'yellow', 'green', 'green', 'blue', 'blue', 'green', 'yellow', 'yellow', 'blue', 'red', 'yellow'], ['blue', 'yellow', 'gree
n', 'blue', 'yellow', 'red', 'blue', 'yellow', 'green', 'yellow', 'yellow', 'green', 'red'], ['green', 'green', 'blue', 'red',
 'red', 'yellow', 'green', 'yellow', 'yellow', 'red', 'blue', 'red', 'red'], ['green', 'green', 'red', 'yellow', 'yellow', 'gr
een', 'yellow', 'red', 'red', 'blue', 'green', 'blue', 'red'], ['green', 'red', 'blue', 'red', 'green', 'yellow', 'yellow', 'b
lue', 'green', 'red', 'yellow', 'green', 'red'], ['yellow', 'red', 'red', 'yellow', 'red', 'blue', 'blue', 'green', 'red', 'ye
llow', 'green', 'yellow', 'red'], ['red', 'blue', 'green', 'green', 'red', 'blue', 'yellow', 'blue', 'green', 'red', 'yellow',
 'red', 'yellow'], ['yellow', 'red', 'blue', 'yellow', 'green', 'red', 'red', 'red', 'red', 'blue', 'green', 'blue', 'red'], [
'blue', 'red', 'green', 'yellow', 'yellow', 'blue', 'yellow', 'red', 'green', 'green', 'green', 'yellow', 'red'], ['red', 'yel
low', 'blue', 'green', 'yellow', 'yellow', 'green', 'yellow', 'red', 'red', 'blue', 'blue', 'green'], ['red', 'green', 'red',
'yellow', 'green', 'green', 'blue', 'blue', 'yellow', 'red', 'red', 'blue', 'green'], ['blue', 'green', 'blue', 'red', 'blue',
 'yellow', 'green', 'blue', 'yellow', 'blue', 'yellow', 'blue', 'red'], ['green', 'yellow', 'yellow', 'yellow', 'green', 'red'
, 'red', 'red', 'blue', 'green', 'red', 'blue', 'yellow']]

Fitness scores [2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
 4, 4, 4, 4, 4, 5, 5, 5]
```

```
Best Parents ['1100011011001101101100100', '1010110111000111100111010', '0001101111001101101010101', '1100011011001101110011001
10101', '0110010011000111100111010', '1000000110111100100111100', '1100011010111011000111010', '1010000101111011110100010'
, '0001101111001101101100010', '0111001011111011010010110', '0100101001001100100111110', '0010111010000010111100111', '001
0110110011001110111100', '0110011111101101010010001', '1100011011001101101111001', '1101001110010101010100100001', '111000101
0111011000111010', '0111001011111101101010000', '0010110000101100001100100', '0100101001001100100111110', '1110011011001001
01110010010', '0011101001010001101100010']

Crossovered ['1100011011000111100111010', '1010110111001101101100010', '0001111011001101110011010', '1100001111001101101010
1101', '0110010011000111100111010', '1000000110111100100111110', '1100011010111011000100010', '1010000101111011110110010',
 '0001101111001101101100101', '0111001011111011001100010', '0100101001001100100100111', '0010111010000010111111110', '0010
1101100110011101111001', '0110011111101101010010010', '1100011011010101010010001', '1101001110001101101111001', '1110001010
1110110001000010', '0111001011111011010110010', '0010110000101100100111110', '0100101001001100001100100', '1110011011001001
1011000100', '0011101001010000111010010']
['1100011011000111100111010', '1010110111001101101100010', '0001111011001101110011010', '1100001111001101101010101101', '0110
0100110001111001110100', '1000000110111100100111110', '1100011010111011000100010', '1010000101111011110110010', '0001101111
0011011100101101', '0111001011111011001100010', '0100101001001100100100111', '0010111010000010111111110', '0010110110011001
1101111001', '0110011111101101010010010', '1100011011010101010010001', '1101001110001101101111001', '1110001010111011000100
0010', '0111001011111011010110010', '0010110000101100100111110', '0100101001001100001100100', '1110011011001001101100010',
 '0011101001010000111010010']
colorsMatrix [['yellow', 'blue', 'red', 'green', 'yellow', 'blue', 'red', 'yellow', 'green', 'red', 'yellow', 'red', 'green'],
 ['green', 'green', 'yellow', 'red', 'yellow', 'blue', 'yellow', 'red', 'green', 'yellow', 'blue', 'red', 'blue'], ['blue', 'r
ed', 'yellow', 'green', 'yellow', 'blue', 'yellow', 'red', 'yellow', 'blue', 'yellow', 'red', 'red'], ['yellow', 'blue', 'blue
', 'yellow', 'yellow', 'blue', 'yellow', 'red', 'green', 'green', 'green', 'yellow', 'red'], ['red', 'green', 'red', 'blue', '
yellow', 'blue', 'red', 'yellow', 'green', 'red', 'yellow', 'red', 'blue'], ['green', 'blue', 'blue', 'red', 'green', 'yellow'
, 'blue', 'green', 'red', 'yellow', 'yellow', 'yellow', 'green'], ['yellow', 'blue', 'red', 'green', 'green', 'yellow', 'green
', 'yellow', 'blue', 'red', 'blue', 'red', 'blue'], ['green', 'green', 'blue', 'red', 'red', 'yellow', 'green', 'yellow', 'yel
low', 'red', 'yellow', 'blue', 'green'], ['blue', 'red', 'green', 'yellow', 'yellow', 'blue', 'yellow', 'red', 'yellow', 'blue
', 'green', 'yellow', 'red'], ['red', 'yellow', 'blue', 'green', 'yellow', 'yellow', 'green', 'yellow', 'blue', 'yellow', 'blu
e', 'blue', 'green'], ['red', 'blue', 'green', 'green', 'red', 'blue', 'yellow', 'blue', 'green', 'red', 'blue', 'red', 'yello
w'], ['blue', 'green', 'yellow', 'green', 'green', 'blue', 'blue', 'green', 'yellow', 'yellow', 'yellow', 'yellow', 'green'],
['blue', 'green', 'yellow', 'red', 'green', 'red', 'green', 'red', 'yellow', 'red', 'yellow', 'green', 'red'], ['red', 'green'
, 'red', 'yellow', 'yellow', 'green', 'yellow', 'red', 'red', 'blue', 'green', 'red', 'blue'], ['yellow', 'blue', 'red', 'gree
n', 'yellow', 'red', 'red', 'red', 'red', 'blue', 'green', 'blue', 'red'], ['yellow', 'red', 'blue', 'yellow', 'green', 'blue'
, 'yellow', 'red', 'green', 'yellow', 'yellow', 'green', 'red'], ['yellow', 'green', 'blue', 'green', 'green', 'yellow', 'gree
n', 'yellow', 'blue', 'red', 'blue', 'blue', 'green'], ['red', 'yellow', 'blue', 'green', 'yellow', 'yellow', 'green', 'yellow
', 'red', 'red', 'yellow', 'blue', 'green'], ['blue', 'green', 'yellow', 'blue', 'blue', 'green', 'yellow', 'blue', 'green', '
red', 'yellow', 'yellow', 'green'], ['red', 'blue', 'green', 'green', 'red', 'blue', 'yellow', 'blue', 'blue', 'yellow', 'gree
n', 'red', 'blue'], ['yellow', 'green', 'red', 'green', 'yellow', 'blue', 'green', 'red', 'green', 'yellow', 'blue', 'red', 'b
lue'], ['blue', 'yellow', 'green', 'green', 'red', 'red', 'blue', 'blue', 'yellow', 'green', 'red', 'blue', 'green']]

Mutated fitness scores [2, 4, 6, 7, 3, 3, 4, 3, 3, 5, 4, 6, 4, 6, 7, 3, 4, 4, 2, 3, 5, 5]
```



Ο αλγόριθμος τερματίζει στην 11η εποχή.

```
Algorithm successfully finished on generation 11 with best coloring being the ['yellow', 'green', 'blue', 'red', 'green', 'yel
low', 'green', 'yellow', 'blue', 'red', 'yellow', 'blue', 'green']
```



Παρατηρούμε πως έχει βρει μία βέλτιστη λύση αφού δεν υπάρχουν γειτονικοί κόμβοι με ίδια χρώματα.

ΒΙΒΛΙΟΓΡΑΦΙΚΕΣ ΑΝΑΦΟΡΕΣ

[1] https://el.wikipedia.org/wiki/Χρωματισμός_ακμών#:~:text=Ο%20χρωματισμός%20ακμών%20είναι%20ένας,όσο%20το%20δυνατόν%20λιγότερα%20χρώματα

(Χρωματισμός ακμών)

[2] https://www.geeksforgeeks.org/project-idea-genetic-algorithms-for-graph-colouring/

(Tutorial)

[3] https://www.youtube.com/watch?v=nhT56blfRpE

(Tutorial)

[4] Διαφάνειες μαθήματος