**­­ A blue and red logo

Description automatically generated**

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 και Έμπειρα Συστήματα, με κύριο στόχο τη δημιουργία ενός γενετικού αλγόριθμου για την επίλυση μίας ειδικής περίπτωσης του προβλήματος χρωματισμού γράφων.

**ΕΚΦΩΝΗΣΗ**

A. 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.

A white rectangular pattern with black numbers

Description automatically generated

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

[ΠΕΡΙΕΧΟΜΕΝΑ 3](#_Toc167918058)

[1. ΕΙΣΑΓΩΓΗ 4](#_Toc167918059)

[1.1 ΣΤΟΧΟΙ ΕΡΓΑΣΙΑΣ 4](#_Toc167918060)

[1.2 ΟΡΙΣΜΟΣ ΤΟΥ ΠΡΟΒΛΗΜΑΤΟΣ ΠΡΟΣ ΕΠΙΛΥΣΗ 4](#_Toc167918061)

[2. ΣΥΝΤΟΜΗ ΠΑΡΟΥΣΙΑΣΗ ΤΟΥ ΠΡΟΒΛΗΜΑΤΟΣ 4](#_Toc167918062)

[3. ΠΕΡΙΓΡΑΦΗ ΠΡΟΓΡΑΜΜΑΤΟΣ 4](#_Toc167918063)

[3.1 ΣΥΛΛΗΨΗ ΑΠΑΙΤΗΣΕΩΝ 4](#_Toc167918064)

[3.2 ΑΝΑΛΥΣΗ ΣΧΕΔΙΑΣΜΟΥ 6](#_Toc167918065)

[3.2.1 Διαγράμματα Περιπτώσεων Χρήσης 6](#_Toc167918066)

[3.2.2 Διαγράμματα Τάξεων 8](#_Toc167918067)

[4. ΕΠΙΔΕΙΞΗ ΤΗΣ ΛΥΣΗΣ 11](#_Toc167918068)

[ΒΙΒΛΙΟΓΡΑΦΙΚΕΣ ΑΝΑΦΟΡΕΣ 11](#_Toc167918069)

# 1. ΕΙΣΑΓΩΓΗ

## 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

H μήτρα είναι μη μηδενική στην θέση Α[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.

## ΓΕΝΙΚΗ ΠΕΡΙΓΡΑΦΗ

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

1. Πλήθος χρωμάτων με τα οποία θα χρωματιστούν οι κόμβοι
2. Πλήθος αρχικού πληθυσμού
3. Πιθανότητα μετάλλαξης χρωμοσώματος/ων σε κάθε γενιά
4. Αριθμός χρωμοσωμάτων που θα μεταλλάσσονται σε κάθε γενιά
5. Μέγιστος αριθμός γενιών
6. 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.

A network with blue dots and lines

Description automatically generated

***Εικόνα 1. Γράφος προβλήματος***

## ΑΝΑΛΥΣΗ ΒΑΣΙΚΩΝ ΣΥΝΑΡΤΗΣΕΩΝ

## generate\_population

1. # generate population
2. def generate\_population(size: int, power: int, dimensions: int): # size for the # of the generated chromosomes
4. population = [] # power for the # of bits needed to represent the colors
5. for i in range(size): # dimensions for the dimensions of the adjacency matrix (# of graph nodes)
6. binary\_number = []
7. for j in range(power\*dimensions):
8. binary\_number.append(str(random.randint(0,1)))
9. population.append(''.join(binary\_number))
10. return population
11. # 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.

## 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.

H 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 = ∑ penalty(i, j)

## tournament\_selection

1. # select best solutions (parents) from the current generation to descend to the following ones
2. def tournament\_selection(population: list, fitness\_score\_array: list, k: int): # k refers to the # of values compared each time
3. indices = [i for i in range(len(fitness\_score\_array))] # will save the indices of the fitness\_score\_array
4. best\_parents = [] # best parents in the current population
5. while len(indices) >= k: # continue until there are enough remaining indices
6. 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)
7. parents = [population[i] for i in parents\_indices] # Get corresponding parents
8. # select fitness scores for the selected parents according to their indices
9. fitness\_scores = [fitness\_score\_array[i] for i in parents\_indices]
10. # sort parents based on fitness scores (search for min)
11. 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]
12. best\_parents.append(sorted\_parents[0])
13. # remove the index of the selected parent from the list of remaining indices
14. for index in parents\_indices:
15. indices.remove(index)
17. # remove the fitness scores of the unselected parents
18. # for index in indices:
19. # fitness\_score\_array.remove(fitness\_score\_array[index])
20. return best\_parents

H 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.

## single\_point\_crossover

1. def single\_point\_crossover(best\_parents: list):
2. i=0
3. # check for same length
4. while i < len(best\_parents) - 1:
5. if len(best\_parents[i]) != len(best\_parents[i+1]):
6. raise ValueError("\nChromosomes must be of the same length")
7. i+=1
9. if len(best\_parents) < 2:
10. return best\_parents
11. crossovered\_parents = []
12. for i in range(0, len(best\_parents) - 1, 2): # iterate through all unique pairs of 2 parents
13. parent1 = best\_parents[i]
14. parent2 = best\_parents[i+1]
15. p = random.randint(1, len(parent1)) # generate a random integer within the bits length
16. crossovered\_parents.append(parent1[0:p]+parent2[p:]) # crossover a
17. crossovered\_parents.append(parent2[0:p]+parent1[p:]) # crossover b
18. 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

## mutate

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

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.

## 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.

1. ΕΠΙΔΕΙΞΗ ΤΗΣ ΛΥΣΗΣ

A screenshot of a computer

Description automatically generatedΓια να αναδείξουμε την λειτουργία του προγράμματος, θα τρέξει ο αλγόριθμος πάνω στο τρέχον πρόβλημα για τις παραμέτρους num\_colors=4, population=100, probability=0.5, num\_mutations=2, generations=10, elite=50.

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

A screen shot of a computer screen

Description automatically generated

A network of colored dots and lines

Description automatically generatedA computer screen with white text

Description automatically generated

Ο αλγόριθμος τερματίζει στην 6η εποχή.



A network with colored circles and lines

Description automatically generated

Παρατηρούμε πως έχει βρει μία βέλτιστη λύση αλλά υπάρχει μία ατέλεια αφού οι κόμβοι 11, 4 παρόλο που γειτονεύουν έχουν το ίδιο χρώμα.

# ΒΙΒΛΙΟΓΡΑΦΙΚΕΣ ΑΝΑΦΟΡΕΣ

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

(Χρωματισμός ακμών)

1. <https://www.geeksforgeeks.org/project-idea-genetic-algorithms-for-graph-colouring/>

(Tutorial)

1. <https://www.youtube.com/watch?v=nhT56blfRpE>

(Tutorial)

1. Διαφάνειες μαθήματος