ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΕΠΙΚΟΙΝΩΝΙΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάθημα Προπτυχιακών Σπουδών:

Τεχνητή Νοημοσύνη και Έμπειρα Συστήματα

Ακαδημαϊκό Έτος: 2022 - 2023

Εξάμηνο: 6ο

Προαιρετική Εργασία

Φοιτητής: Θεόδωρος Κοξάνογλου

Α.Μ: Π20094

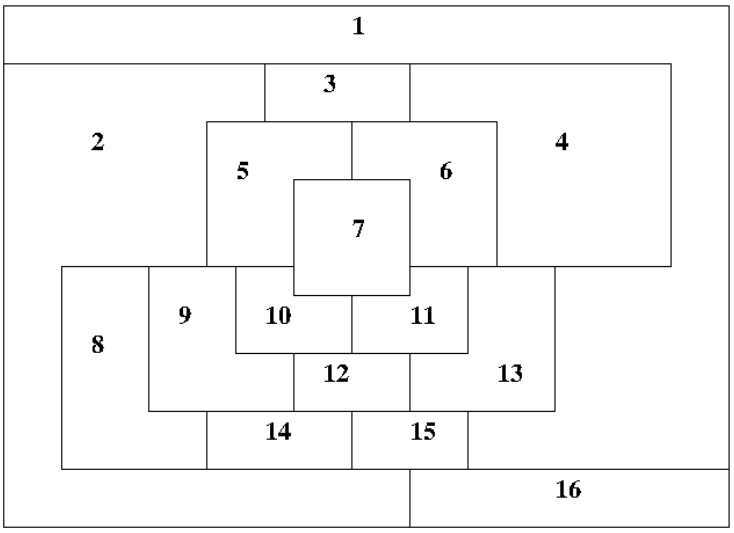
Περιεχόμενα

Εκφώνηση

Θέμα προαιρετικής εργασίας για το μάθημα «Τεχνητή Νοημοσύνη και Έμπειρα Συστήματα». Bonus 2 βαθμοί.

Η εργασία είναι ατομική. Παραδοτέο είναι αρχείο pdf με τεκμηριωμένο κώδικα και παραδείγματα εκτέλεσης, περίπου 5 σελίδων. Υποβολή αποκλειστικά μέσω gune2. ΜΗ ΣΤΕΙΛΕΤΕ EMAIL. ΜΗ ΣΤΕΙΛΕΤΕ ΠΗΓΑΙΟ ΚΩΔΙΚΑ.

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 καταλληλότητας 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.

Παραδοτέα της εργασίας είναι ένα pdf (όχι zip, όχι πηγαίος κώδικας) που να περιλαμβάνει τον κώδικα και να τον εξηγεί, να εξηγεί τον τρόπο δράσης του υπολογιστή σύμφωνα με τον αλγόριθμο επίλυσης και να περιλαμβάνει παραδείγματα εκτέλεσης του προγράμματος που αναπτύξατε.

Πηγαίος Κώδικας

```
import random
import networkx as nx
from matplotlib import pyplot as plt

graph = {
    1: [2, 3, 4, 13, 15, 16],
    2: [1, 3, 5, 8, 9, 14, 15, 16],
    3: [1, 2, 4, 5, 6],
    4: [1, 3, 6, 13],
    5: [2, 3, 6, 7, 9, 10],
    6: [3, 4, 5, 7, 11, 13],
    7: [5, 6, 10, 11],
    8: [2, 9, 14],
    9: [2, 5, 8, 10, 12, 14],
    10: [5, 7, 9, 11, 12],
    11: [6, 7, 10, 12, 13],
    12: [9, 10, 11, 13, 14, 15],
    13: [1, 4, 6, 11, 12, 15],
    14: [2, 8, 9, 12, 15],
    15: [1, 2, 12, 13, 14, 16],
    16: [1, 2, 15]
}
colors = ["Red", "Green", "Blue", "Yellow"]


def create_population():
    population_list = []
    for k in range(population_size):
        individual_dict = {node: random.choice(colors) for node in graph.keys()}
        population_list.append(individual_dict)
    print("Population created successfully!")
    return population_list


def fitness(individual_dict):
    fitness_counter = 0
    for node in graph:
        for neighbor in graph[node]:
            if individual_dict[node] != individual_dict[neighbor]:
                fitness_counter += 1
    print("Fitness score for an individual is:", int(fitness_counter / 2))
    return int(fitness_counter / 2)  # Because we are counting each edge twice


def fitness_score_per_individual(population_list):
    fitness_scores = {}
    for k, individual_dict in enumerate(population_list):
        fitness_scores[k] = fitness(individual_dict)
    return fitness_scores


# We are going to select the parents using roulette wheel selection
def select_parents(population_list, fitness_scores):
    # Calculate the sum of all fitness scores
    sum_of_fitness_scores = sum(fitness_scores)
    # Calculate the probability of each individual
    probabilities = [fitness_score_count / sum_of_fitness_scores for fitness_score_count in
                     fitness_scores]  # fitness_score / sum_of_fitness_scores = probability
    # Calculate the cumulative probability of each individual
    cumulative_probabilities = [sum(probabilities[:k + 1]) for k in range(len(probabilities))]
    # Select two parents
    parents_list = []
    for k in range(2):
        random_number = random.random()
        for (index, individual_dict) in enumerate(population_list):
            if random_number <= cumulative_probabilities[index]:
                parents_list.append(individual_dict)
                break
    if len(parents_list) != 2:
        print("Error in selecting parents")
    return parents_list


def crossover(parent1, parent2):
    crossover_point = random.randint(1, num_of_nodes - 1)  # We don't want to include the last node
    child1 = {}
    child2 = {}
    for k, node in enumerate(graph.keys()):
        if k <= crossover_point:
            child1[node] = parent1[node]
            child2[node] = parent2[node]
        else:
            child1[node] = parent2[node]
            child2[node] = parent1[node]
    return child1, child2
```

```
def mutation(child1, child2):
    child = random.choice([child1, child2])
    for node in graph:
        for neighbor in graph[node]:
            if child[node] == child[neighbor]:
                colors_temp = colors.copy()
                colors_temp.remove(child[node])
                child[node] = random.choice(colors_temp)  # We don't want to have the same color as the
neighbor
                break
    return child1, child2


def create_children(parent1, parent2):
    child1 = {}
    child2 = {}
    random_number = random.randint(0, 2)
    if random_number == 0:
        child1, child2 = crossover(parent1, parent2)
    elif random_number == 1:
        child1, child2 = mutation(parent1, parent2)
    elif random_number == 2:
        child1, child2 = crossover(parent1, parent2)
        child1, child2 = mutation(child1, child2)
    return child1, child2


if __name__ == '__main__':
    print("Genetic Algorithm for coloring graph")
    print("Created by: Theodoros Koxanoglou, AM: P20094")
    parents = []
    mates = []
    children = []
    num_of_nodes = len(graph)
    print("The number of nodes in the graph is:", num_of_nodes)
    population_size = # complete the size
    print("We manually set the population size to", population_size, "individuals")
    # Create population
    population = create_population()
    # Calculate fitness
    fitness_scores_dict = fitness_score_per_individual(population)

    correct_fitness_score = 42
    for i in range(# complete the number of population):
        print("-------------------------Generation:", i + 1, "----------------------------")
        # Select parents
        for j in range(len(population) // 2):
            mates.append(select_parents(population, fitness_scores_dict))
        # Create children
        for j in range(len(mates)):
            children.extend(
                create_children(mates[j][0], mates[j][1]))  # We need to create two children for each pair
of parents

        population = children  # We need to replace the old population with the new one
        children = []
        mates = []
        parents = []

        # Calculate fitness
        fitness_scores_dict = fitness_score_per_individual(population)
        if correct_fitness_score in fitness_scores_dict.values():
            print("We found a solution!")
            break
    sorted_fitness_scores_dict = dict(sorted(fitness_scores_dict.items(), key=lambda x: x[1]))
    solution = list(sorted_fitness_scores_dict.keys())[-1]
    if fitness_scores_dict[solution] == correct_fitness_score:
        print("The solution is:", population[solution])
    else:
        print("The optimal solution is:", population[solution], "with fitness score:", solution,
              "and percentage:", "{:.2f}%".format(solution / correct_fitness_score * 100))
    G = nx.Graph()
    G.add_nodes_from(graph.keys())
    for node in graph:
        for neighbor in graph[node]:
            G.add_edge(node, neighbor)
    nx.draw(G, with_labels=True, font_weight='bold', node_color=list(population[solution].values()))
    plt.show()
    print("Thank you for using the Genetic Algorithm for coloring graph")
```

Ανάλυση Προγράμματος

Το πρόγραμμα δημιουργήθηκε σε γλώσσα προγραμματισμού Python.

Στην αρχή του προγράμματος όρισα τους γείτονες των κόμβων του σχήματος σε ένα λεξικό '**graph**' και τα διαθέσιμα χρώματα σε μία λίστα '**colors'**:

```
graph = {
    1: [2, 3, 4, 13, 15, 16],
    2: [1, 3, 5, 8, 9, 14, 15, 16],
    3: [1, 2, 4, 5, 6],
    4: [1, 3, 6, 13],
    5: [2, 3, 6, 7, 9, 10],
    6: [3, 4, 5, 7, 11, 13],
    7: [5, 6, 10, 11],
    8: [2, 9, 14],
    9: [2, 5, 8, 10, 12, 14],
    10: [5, 7, 9, 11, 12],
    11: [6, 7, 10, 12, 13],
    12: [9, 10, 11, 13, 14, 15],
    13: [1, 4, 6, 11, 12, 15],
    14: [2, 8, 9, 12, 15],
    15: [1, 2, 12, 13, 14, 16],
    16: [1, 2, 15]
}
colors = ["R", "G", "B", "Y"]
```

Βήματα Γενετικού Αλγορίθμου:

Βήμα 1:

Στην αρχή όρισα το μέγεθος του πληθυσμού και δημιούργησα έναν τυχαίο αρχικό πληθυσμό μέσω της μεθόδου **def create_population()**:

Κώδικας από την main συνάρτηση:

```
if __name__ == '__main__':
    print("Genetic Algorithm for coloring graph")
    print("Created by: Theodoros Koxanoglou, AM: P20094")
    parents = []
    mates = []
    children = []
    num_of_nodes = len(graph)
    print("The number of nodes in the graph is:", num_of_nodes)
    population_size = 10
    print("We manually set the population size to", population_size, "individuals")
    # Create population
    population = create_population()
```

Κώδικας της **def create_population()**:

```
1 usage
def create_population():
    population_list = []
    for k in range(population_size):
        individual_dict = {node: random.choice(colors) for node in graph.keys()}
        population_list.append(individual_dict)
    print("Population created successfully!")
    return population_list
```

Βήμα 2:

Έφτιαξα μία συνάρτηση καταλληλότητας **def fitness(individual_dict)** και υπολόγισα την καταλληλότητα όλων των υποψήφιων από την αρχικό πληθυσμό.

Κώδικας από την main συνάρτηση:

```
    # Calculate fitness
    fitness_scores_dict = fitness_score_per_individual(population)
```

Συνάρτηση **def fitness_score_per_individual(population_list)** που επιστρέφει το '**fitness_score**' όλων των individuals του πληθυσμού:

```
2 usages
def fitness_score_per_individual(population_list):
    fitness_scores = {}
    for k, individual_dict in enumerate(population_list):
        fitness_scores[k] = fitness(individual_dict)
    return fitness_scores
```

Συνάρτηση **def fitness(individual_dict)** για τον υπολογισμό του '**fitness_score**' ενός individual:

```
def fitness(individual_dict):
    fitness_counter = 0
    for node in graph:
        for neighbor in graph[node]:
            if individual_dict[node] != individual_dict[neighbor]:
                fitness_counter += 1
    print("Fitness score for an individual is:", int(fitness_counter / 2))
    return int(fitness_counter / 2)  # Because we are counting each edge twice
```

Βήμα 3:

Επιλογή γονέων για αναπαραγωγή. Σχηματίζω ζεύγη individuals των οποίων το πλήθος τους εξαρτάται από τον τύπο: **individuals**/2 **parents** = y **mates**, δίνοντας μεγαλύτερη προτεραιότητα στο ζευγάρωμα γονέων με μεγαλύτερο '**fitness_score**' χρησιμοποιώντας την τεχνική της ρουλέτας. Η τεχνική της ρουλέτας είναι:

1) Υπολογίζω στην αρχή το άθροισμα όλων των τιμών καταλληλότητας των υποψήφιων λύσεων.
2) Επιλέγω έναν τυχαίο αριθμό από το 0 μέχρι το 1 χρησιμοποιώντας την συνάρτηση ομοιόμορφης κατανομής αριθμών: **random.random()**
3) Εξετάζω κάθε πιθανό γονέα αν το **fitness_score** του είναι μεγαλύτερο ή ίσο με τον τυχαίο αριθμό που δημιούργησε η **random.random()**. Αν είναι μεγαλύτερο το ποσοστό προστίθεται ο γονέας στη λίστα parents_list.
4) Επαναλαμβάνεται η διαδικασία 3) για τον δεύτερο γονέα. Όταν επιλεγεί **και** ο δεύτερος γονέας η επανάληψη της τεχνικής της ρουλέτας τερματίζεται και επιστρέφει μία λίστα με τους δύο γονείς.

Κώδικας από την main συνάρτηση:

```
        # Select parents
        for j in range(len(population) // 2):
            mates.append(select_parents(population, fitness_scores_dict))
```

Συνάρτηση **def select_parents(population_list, fitness_scores)** επιλογής γονέων:

```python
# We are going to select the parents using roulette wheel selection
1 usage
def select_parents(population_list, fitness_scores):
    # Calculate the sum of all fitness scores
    sum_of_fitness_scores = sum(fitness_scores)
    # Calculate the probability of each individual
    probabilities = [fitness_score_count / sum_of_fitness_scores for fitness_score_count in
                     fitness_scores]  # fitness_score / sum_of_fitness_scores = probability
    # Calculate the cumulative probability of each individual
    cumulative_probabilities = [sum(probabilities[:k + 1]) for k in range(len(probabilities))]
    # Select two parents
    parents_list = []
    for k in range(2):
        random_number = random.random()
        for (index, individual_dict) in enumerate(population_list):
            if random_number <= cumulative_probabilities[index]:
                parents_list.append(individual_dict)
                break
    if len(parents_list) != 2:
        print("Error in selecting parents")
    return parents_list
```

Βήμα 4:

Όταν τελειώσει η επιλογή όλων των γονέων, θα ξεκινήσει η αναπαραγωγή. Θα δημιουργηθεί η νέα γενιά βασισμένη στα παιδιά των γονέων που επιλέχθηκαν στο προηγούμενο βήμα. Οι τελεστές που χρησιμοποιήθηκαν για την αναπαραγωγή του πληθυσμού είναι κυρίως 3:

1) Ομοιόμορφη διασταύρωση
2) Μετάλλαξη σημείου
3) Ομοιόμορφη διασταύρωση και Μετάλλαξη σημείου

Και οι τρεις τελεστές μπορούν να «ενεργοποιηθούν» με πιθανότητα **1/3**.

Κώδικας από την main συνάρτηση:

```python
        # Create children
        for j in range(len(mates)):
            children.extend(
                create_children(mates[j][0], mates[j][1]))  # We need to create two children for each pair of parents
```

Συνάρτηση **def create_children(parent1, parent2)** αναπαραγωγής παιδιών:

```python
2 usages
def mutation(child1, child2):
    child = random.choice([child1, child2])
    for node in graph:
        for neighbor in graph[node]:
            if child[node] == child[neighbor]:
                colors_temp = colors.copy()
                colors_temp.remove(child[node])
                child[node] = random.choice(colors_temp)  # We don't want to have the same color as the neighbor
                break
    return child1, child2
```

Για την ομοιόμορφη διασταύρωση **def crossover(parent1, parent2)**:

```python
def crossover(parent1, parent2):
    crossover_point = random.randint(1, num_of_nodes - 1)  # We don't want to include the last node
    child1 = {}
    child2 = {}
    for k, node in enumerate(graph.keys()):
        if k <= crossover_point:
            child1[node] = parent1[node]
            child2[node] = parent2[node]
        else:
            child1[node] = parent2[node]
            child2[node] = parent1[node]
    return child1, child2
```

Για την μετάλλαξη σημείου/κόμβου σε ένα child individual **def mutation(child1, child2)**:

```python
2 usages
def mutation(child1, child2):
    child = random.choice([child1, child2])
    for node in graph:
        for neighbor in graph[node]:
            if child[node] == child[neighbor]:
                colors_temp = colors.copy()
                colors_temp.remove(child[node])
                child[node] = random.choice(colors_temp) # We don't want to have the same color as the neighbor
                break
    print("Child after mutation:", child)
    return child1, child2
```

Ο νέος πληθυσμός αποτελείται από το σύνολο των απογόνων οι οποίοι, εν μέρη, θα έχουν καλύτερο ποσοστό καταλληλότητας από τους προγόνους τους.

Βήμα 5:

Υπολογίζεται εκ νέου το ποσοστό καταλληλότητας του νέου πληθυσμού.

Από την βασική συνάρτηση main:

```python
        fitness_scores_dict = fitness_score_per_individual(population)
        if correct_fitness_score in fitness_scores_dict.values():
            print("We found a solution!")
            break
```

Βήμα 6:

Αν δεν υπάρχει κάποια υποψήφια λύση, επαναλαμβάνεται η διαδικασία από το βήμα **3)** μέχρι:

- Είτε να βρεθεί μία τουλάχιστον λύση
- Είτε να τελειώσει η αναπαραγωγή της n-στης γενιάς (στον κώδικα έχω ορίσει 100 γενιές).

Από την βασική συνάρτηση main:

```
correct_fitness_score = 42
for i in range(100):
    print("------------------------------Generation:", i + 1, "------------------------------")
    # Select parents
    for j in range(len(population) // 2):
        mates.append(select_parents(population, fitness_scores_dict))
    # Create children
    for j in range(len(mates)):
        children.extend(
            create_children(mates[j][0], mates[j][1]))  # We need to create two children for each pair of parents

    population = children  # We need to replace the old population with the new one
    children = []
    mates = []
    parents = []

    # Calculate fitness
    fitness_scores_dict = fitness_score_per_individual(population)
    if correct_fitness_score in fitness_scores_dict.values():
        print("We found a solution!")
        break
```

Ανεξαρτήτου αποτελέσματος στο τέλος του προγράμματος, θα εμφανίζεται στην οθόνη του χρήστη ένα ενδεικτικό σχήμα με τους γειτονικούς κόμβους και τα αντίστοιχα χρώματά τους. Αν το πρόγραμμα βρει τουλάχιστον μία λύση θα εμφανιστεί στον χρήστη ένα μήνυμα επιτυχίας μαζί με την λύση που βρήκε το πρόγραμμα. Αν το πρόγραμμα δεν βρει τουλάχιστον μία λύση, θα εμφανιστεί στον χρήστη η βέλτιστη λύση της τελευταίας γενιάς μαζί με το ποσοστό fitness_score της λύσης αυτής.

Στιγμιότυπα εκτέλεσης Κώδικα:

Όταν εκτελεστεί το πρόγραμμα θα εκτυπωθούν στην οθόνη τα ακόλουθα μηνύματα:

```
Genetic Algorithm for coloring graph
Created by: Theodoros Koxanoglou, AM: P20094
The number of nodes in the graph is: 16
We manually set the population size to 100 individuals
Population created successfully!
```

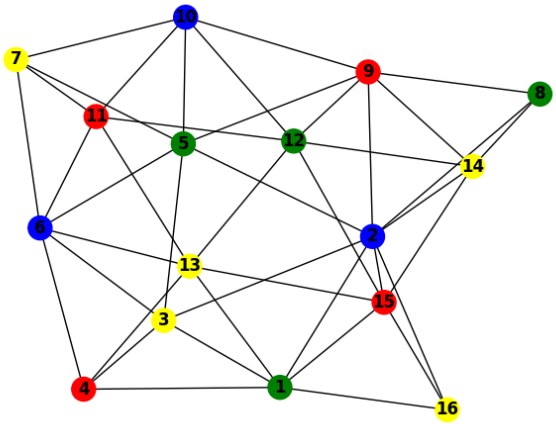
Στο τέλος κάθε γενιάς θα εκτυπώνεται ένα σχετικό μήνυμα για την αλλαγή της γενιάς:

```
Fitness score for an individual is: 32
Fitness score for an individual is: 30
-----------------------------Generation: 1 -----------------------------
Fitness score for an individual is: 36
Fitness score for an individual is: 33
```

```
Fitness score for an individual is: 29
-----------------------------Generation: 2 -----------------------------
Fitness score for an individual is: 33
Fitness score for an individual is: 35
Fitness score for an individual is: 31
Fitness score for an individual is: 34
```

Αν βρεθεί λύση:

```
We found a solution!
The solution is: {1: 'Green', 2: 'Blue', 3: 'Yellow', 4: 'Red', 5: 'Green', 6: 'Blue', 7: 'Yellow', 8: 'Green', 9: 'Red', 10: 'Blue', 11: 'Red', 12: 'Green', 13: 'Yellow', 14: 'Yellow', 15: 'Red', 16: 'Yellow'}
Thank you for using the Genetic Algorithm for coloring graph
```



Αν δεν βρεθεί λύση, θα εκτυπωθεί η βέλτιστη λύση της προηγούμενης γενιάς μαζί με το αντίστοιχο ποσοστό fitness_score που θα έχει:

```
The optimal solution is: {1: 'Yellow', 2: 'Blue', 3: 'Red', 4: 'Blue', 5: 'Green', 6: 'Yellow', 7: 'Yellow', 8: 'Green', 9: 'Red', 10: 'Blue', 11: 'Green', 12: 'Yellow', 13: 'Red', 14: 'Red', 15: 'Green', 16: 'Blue'} with fitness score: 5 and percentage: 11.90%
Thank you for using the Genetic Algorithm for coloring graph
```