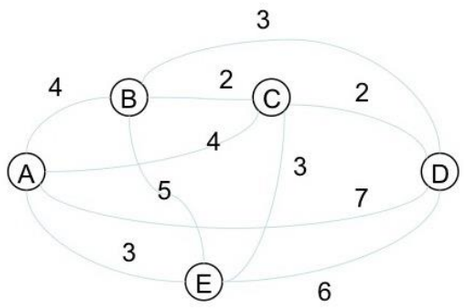
Εργασία Μαθήματος «ΤΕΧΝΗΤΗ ΝΟΗΜΟΣΥΝΗ»

Όνομα φοιτητή – Αρ. Μητρώου	**ΕΥΘΥΜΙΟΣ-ΠΑΤΡΟΚΛΟΣ ΓΕΩΡΓΙΑΔΗΣ – Π19031**
Προθεσμία παράδοσης	06/06/2022

Περιεχόμενα

Κύριο Θέμα

Α. Αναπτύξτε πρόγραμμα επίλυσης του Traveling Salesman Problem με χρήση γενετικών αλγορίθμων και γλώσσα προγραμματισμού της επιλογής σας. Ο γράφος αποτελείται από πλήρως διασυνδεδεμένες πόλεις όπως φαίνεται στο παρακάτω σχήμα.



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 μία σύντομη αναφορά που να περιλαμβάνει τον τρόπο δράσης του υπολογιστή σύμφωνα με τον αλγόριθμο επίλυσης και παραδείγματα εκτέλεσης του προγράμματος που αναπτύξατε.

Κώδικας Προγράμματος

- *main.py*

```
import random
from cost_calculator import CostCalculator
from reproduction import Reproduction
from itertools import groupby


def city_shuffle():
    """
    This method is used to randomly create and return a new possible
trip
    """
    path = "A"
    cities = ["B", "C", "D", "E"]
    random.shuffle(cities)
```

```
    i = 0
    while i < len(cities):
        path += cities[i]
        i += 1
    path += "A"
    return path


def parent_picker(self, totalS, idx, path):
    """
    This method is used to identify the randomly chosen member of the
population as a parent
    Args:
        self: The randomly chosen member, represented by a float
number
        totalS: The total sum of the chances of picking each member
of the current population
        idx: The amount of members of the new population
        path: The table including the members of the current
population
    """
    while idx >= 0:
        totalS -= CostCalculator.path_cost(path[idx])
        if totalS <= self:
            return path[idx], idx
        else:
            idx -= 1


def all_equal(self):
    """
    This method is used to calculate if the current population
consists of the same member
    (self is the table of all members of the current population)
    """
    res = groupby(self)
    return next(res, True) and not next(res, False)


def mutate(self):
    """
    This method is used to mutate a member
    """
    # find 2 random cities that aren't the first or the last one
    mutated_spots = random.sample(range(4), 2)
    mutated = [self[0], self[1], self[2], self[3], self[4], self[5]]
    # switch the order of these 2 cities
    temp = self[int(mutated_spots[0]+1)]
    mutated[int(mutated_spots[0] + 1)] = self[int(mutated_spots[1] +
1)]
    mutated[int(mutated_spots[1] + 1)] = temp
    return mutated


# creating the original population (amount N)
paths = []
for N in range(10):
    paths.append(city_shuffle())

# main algorithm
generations = 0
pathsN = len(paths)
# convergence process
while not(all_equal(paths)):
    child_counter = 0
    totalChance = 0
    print("Generation " + str(generations) + ": " + str(paths))

    # calculating the cost of each path
    for og_population_idx in range(pathsN):
        print(CostCalculator.path_cost(paths[og_population_idx]))
        totalChance +=
CostCalculator.path_cost(paths[og_population_idx])
    print(totalChance)

    # picking randomly the new parents
```

```
    random_samples = [0]*round(pathsN * 4 / 5)
    for j in range(0, round(pathsN * 4 / 5), 2):
        random_samples[j] = random.uniform(0, totalChance)
        random_samples[j], p1idx = parent_picker(random_samples[j],
totalChance, og_population_idx, paths)
        while True:
            random_samples[j + 1] = random.uniform(0, totalChance)
            random_samples[j + 1], p2idx =
parent_picker(random_samples[j+1], totalChance, og_population_idx,
paths)
            # making sure they are not the same person
            if p1idx != p2idx:
                break
    print("Parents:"+str(random_samples))

    # making the new generation
    # adding some old chromosomes to the new generation (20% of the
new generation)
    for idx in range(0, round(pathsN * 1 / 5)):
        paths[idx] = paths[random.randint(0, pathsN - 1)]
    # adding the children (80% of the new population)
    for child_idx in range(round(pathsN * 1 / 5), pathsN, 2):
        # adding the children of each "couple"
        paths[child_idx] =
"".join(Reproduction.generate_child(random_samples[child_counter],
random_samples[child_counter+1]))
        paths[child_idx+1] =
"".join(Reproduction.generate_child(random_samples[child_counter+1],
random_samples[child_counter]))
        child_counter += 2

    # mutation process (10% chance)
    for mut in range(pathsN):
        chance = random.randint(1, 10)
        if chance == 1:
            print("Pre-Mutation: " + str(paths[mut]))
            paths[mut] = "".join(mutate(paths[mut]))
            print("Post-Mutation: " + str(paths[mut]))


    generations += 1
    print("Children:"+str(paths))
```

- *reproduction.py*

```
class Reproduction:

    def generate_child(self, partner):
        """
        This method is used to generate the child of two members of
the current population and return it
        Args:
            self: The first parent
            partner: The second parent
        """
        # single point crossover (2 cities from 1st parent, 2 cities
from second)

        # all the cities that will be used from the 1st parent
        visited = [self[0], self[1], self[2], self[5]]
        # all the cities that will be used, if not already, from the
2nd parent
        partner_cities = [partner[3], partner[4], partner[1],
partner[2]]
        # repeat until all cities of the 2nd parent are implemented
in the child trip
        for x in range(len(partner_cities)):
            # if the city is already in the child trip, skip
            if partner_cities[x] in visited:
                continue
            else:
                # the first time a city from the 2nd parent isn't
included, it is inserted as the 4th city in the child
                if len(visited) == 4:
```

```
                    visited.insert(3, partner_cities[x])
                # the second time a city from the 2nd parent isn't
included, it is inserted as the 5th
                else:
                    visited.insert(4, partner_cities[x])
        return visited
```

- *cost_calculator.py*

```
# all possible paths and their costs
pathsdict = {
        "AB": 4,
        "AC": 4,
        "AD": 7,
        "AE": 3,
        "BC": 2,
        "BD": 3,
        "BE": 5,
        "CD": 2,
        "CE": 3,
        "DE": 6
    }


class CostCalculator:

    def path_cost(self):
        """
        This method is used to calculate and return the cost of an
entire trip, from start to end.
        """
        i = 0
        cost: int = 0
        # until the algorithm crosses all of the paths between the
cities on this trip
        while i < len(self) - 2:
            try:
                # search the dictionary pathsdirect, to find the
appropriate cost of this path
                path = self[i] + self[i+1]
                cost += pathsdict[path]
            except:
                # if the path can't be found, reverse the order of
the cities
                path = self[i + 1] + self[i]
                cost += pathsdict[path]
            i += 1

        return 10/cost
```

Ανάλυση Κώδικα

Το πρόγραμμα χωρίζεται σε 3 αρχεία python (.py):

- main.py, το κύριο αρχείο που θα εκτελέσουμε
- reproduction.py, το αρχείο με την κλάση Reproduction, που είναι υπεύθυνη για την δημιουργία απογόνων
- cost_calculator.py, το αρχείο με την κλάση CostCalculator, που είναι υπεύθυνη για την κοστολόγηση των τιμών του κάθε μονοπατιού

- *main.py*

Για την εκτέλεση του προγράμματος μας, θα χρειαστεί να εισάγουμε μερικές μεθόδους:

1. random, για την δημιουργία μεθόδων που εξάγουν τυχαία αποτελέσματα
2. CostCalculator, για την κοστολόγηση των τιμών του κάθε μονοπατιού
3. Reproduction, για την δημιουργία των απογόνων
4. groupby από το itertools, για την εξακρίβωση αν όλα τα μέλη του πληθυσμού είναι ίδια

Θα χρειαστεί επίσης να δημιουργήσουμε μερικές μεθόδους:

1. *city_shuffle()*
   Είσοδος: τίποτα
   Έξοδος: path (μεταβλητή string)

   Η μέθοδος αυτή δημιουργεί μία μεταβλητή path (string), με τιμή “A”, όπου θα προσθέσει τις τιμές string “B” ,“C”, “D”, “E” με τυχαία σειρά και στο τέλος θα ξαναπροσθέσει την τιμή “A”, δημιουργώντας έτσι ένα τυχαίο μονοπάτι ενός πλανόδιου πωλητή και θα επιστρέψει το αποτελέσμα.

```
def city_shuffle():
    """
    This method is used to randomly create and return a new
possible trip
    """
    path = "A"
    cities = ["B", "C", "D", "E"]
    random.shuffle(cities)
    i = 0
    while i < len(cities):
        path += cities[i]
        i += 1
    path += "A"
    return path
```

2. parent_picker(self, totalS, idx, path)
   Είσοδος: self (μεταβλητή float): ένα μέλος του τωρινού πληθυσμού, που επιλέχτηκε ως γονέας, που αντιπροσωπεύεται από μία τυχαία τιμή τύπου float
   totalS (μεταβλητή float): το συνολικό άθροισμα των πιθανοτήτων επιλογής κάθε μονοπατιού του τωρινού πληθυσμού.
   idx (μεταβλητή int): το πλήθος των μελών του επόμενου πληθυσμού
   path (μεταβλητή τύπου array/list): ο πίνακας με τα μέλη του τωρινού πληθυσμού
   Έξοδος: path[idx] (μεταβλητή string): το μέλος του τωρινού πληθυσμού, που επιλέχτηκε ως γονέας
   idx (μεταβλητή int): η θέση του μέλους-γονέα, στον πίνακα του τωρινού πληθυσμού

   Η μέθοδος αυτή χρησιμοποιείται για την αναγνώριση του επιλεγμένου γονέα, χρησιμοποιώντας την πιθανότητα επιλογής του. Ουσιαστικά, αφαιρούμε από το συνολικό άθροισμα των πιθανοτήτων επιλογής κάθε μονοπατιού, μία-μία την κάθε πιθανότητα, μέχρι να γίνει μικρότερο από την τιμή που αντιπροσωπεύει τον επιλεγμένο γονέα. Τότε, επιστρέφουμε το μέλος του τωρινού πληθυσμού, που επιλέχτηκε ως γονέας και την θέση του.

```
def parent_picker(self, totalS, idx, path):
    """
    This method is used to identify the randomly chosen member of
the population as a parent
    Args:
        self: The randomly chosen member, represented by a float
number
        totalS: The total sum of the chances of picking each
member of the current population
        idx: The amount of members of the new population
        path: The table including the members of the current
population
    """
    while idx >= 0:
        totalS -= CostCalculator.path_cost(path[idx])
        if totalS <= self:
            return path[idx], idx
        else:
            idx -= 1
```

3. all_equal(self):
   Είσοδος: self (μεταβλητή τύπου array/list): ο πίνακας με τα μέλη του τωρινού πληθυσμού
   Έξοδος: True ή False (μεταβλητή Boolean)

   Η μέθοδος αυτή ελέγχει εάν τα μέλη του τωρινού πληθυσμού είναι όλα ίδια, δηλαδή εάν ο πληθυσμός συγκλίνει. Εάν ναι, επιστρέφει True, αλλιώς False.

```
def all_equal(self):
    """
    This method is used to calculate if the current population
consists of the same member
    (self is the table of all members of the current population)
    """
    res = groupby(self)
    return next(res, True) and not next(res, False)
```

4. mutate(self):
   Είσοδος: self (μεταβλητή string): ένα μέλος του τωρινού πληθυσμού
   Έξοδος: mutated (μεταβλητή string): το μέλος αφού υποστεί μετάλλαξη

   Η μέθοδος αυτή χρησιμοποιείται για την μετάλλαξη ενός μέλους. Επιλέγουμε 2 τυχαίες πόλεις του μονοπατιού του μέλους και αντιστρέφουμε τις θέσεις τους στο μονοπάτι. Ύστερα, επιστρέφουμε το αποτέλεσμα.

```
def mutate(self):
    """
    This method is used to mutate a member
    """
    # find 2 random cities that aren't the first or the last one
    mutated_spots = random.sample(range(4), 2)
    mutated = [self[0], self[1], self[2], self[3], self[4], self[5]]
    # switch the order of these 2 cities
    temp = self[int(mutated_spots[0]+1)]
    mutated[int(mutated_spots[0] + 1)] = self[int(mutated_spots[1] +
1)]
    mutated[int(mutated_spots[1] + 1)] = temp
    return mutated
```

Τέλος θα χρησιμοποιήσουμε τις μεθόδους generate_child(self, partner), path_cost(self) από τις κλάσεις Reproduction και CostCalculator αντίστοιχα. (αναλυτικότερα πιο κάτω)

Για να εκκινήσει ο αλγόριθμος, θα πρέπει να αρχικοποιήσουμε μερικές μεταβλητές, καθώς και να δημιουργήσουμε τον αρχικό πληθυσμό. Έστω ότι ο αρχικός πληθυσμός και οι επόμενοι πληθυσμοί, έχουν 10 μέλη. Για N=10, θα εκτελέσουμε την city_shuffle() και θα προσθέσουμε στον πληθυσμό με το όνομα paths, 10 τυχαία μέλη ως αρχικό πληθυσμό. Η αρχική γενιά, έχει όνομα “generation 0”, ενώ οι επόμενες θα έχουν αντίστοιχο όνομα, συμβολίζοντας ποια γενιά είναι (π.χ. η $1^{η}$ καινούργια γενιά θα είναι η “generation 1”, η $6^{η}$ θα είναι η “generation 6”, κλπ).

```
# creating the original population (amount N)
paths = []
for N in range(10):
    paths.append(city_shuffle())

# main algorithm
generations = 0
pathsN = len(paths)
```

Τώρα ο γενετικός αλγόριθμος, μπορεί να εκκινήσει κανονικά. Όσο τα μέλη του τωρινού πληθυσμού δεν συγκλίνουν (ελέγχουμε με την μέθοδο all_equal()), τότε ο αλγόριθμος συνεχίζεται και επαναλαμβάνεται. Στην αρχή κάθε επανάληψης, μηδενίζουμε 2 μεταβλητές: την child_counter (int), δηλαδή τον αριθμό των παιδιών

της τωρινής γενιάς, και την totalChance (float), το συνολικό άθροισμα των πιθανοτήτων επιλογής κάθε μονοπατιού του τωρινού πληθυσμού.

```
while not(all_equal(paths)):
    child_counter = 0
    totalChance = 0
```

Για όλα τα μέλη του τωρινού πληθυσμού, θα υπολογίσουμε το αντίστοιχο κόστος της διαδρομής του μέλους με την μέθοδο path_cost() της κλάσης CostCalculator (λεπτομέρειες πιο κάτω). Θα προσθέσουμε το κάθε κόστος στην μεταβλητή totalChance.

```
for og_population_idx in range(pathsN):
    print(CostCalculator.path_cost(paths[og_population_idx]))
    totalChance += CostCalculator.path_cost(paths[og_population_idx])
print(totalChance)
```

Διαδικασία επιλογής γονέων:
Επιλέγουμε να κάνουμε μερική ανανέωση πληθυσμού, με το 20% του τωρινού πληθυσμού να συνυπάρχει με τον επόμενο πληθυσμό. Για να το πετύχουμε αυτό, θα δημιουργήσουμε τον πίνακα random_samples, ο οποίος περιέχει όσα μέλη όσο είναι το 80% του τωρινού πληθυσμού (σε κάθε επανάληψη δηλαδή, 8 μέλη). Τα μέλη αυτά θα είναι αρχικά μηδενικά.

Επίσης θα χρησιμοποιήσουμε την τεχνική της ρουλέτας. Για κάθε 2 μέλη αυτού του πίνακα random_samples θα διαλέξουμε έναν τυχαίο ρητό αριθμό (float) από το 0 έως το totalChance (το συνολικό άθροισμα των πιθανοτήτων επιλογής κάθε μονοπατιού του τωρινού πληθυσμού). Αυτός ο αριθμός θα αντιπροσωπεύει ένα από τα 10 μέλη του τωρινού πληθυσμού. Μέσω της μεθόδου parent_picker(), βρίσκουμε το ακριβές μέλος καθώς και την θέση του στον πίνακα του τωρινού πληθυσμού. Αυτός θα είναι ο πρώτος γονέας του αντίστοιχου ζευγαριού. Έπειτα, επαναλαμβάνουμε την διαδικασία επιλογής τυχαίου μέλους για γονέα. Εάν οι γονείς είναι το ίδιο ακριβώς μέλος (όχι εάν περιέχουν την ίδια διαδρομή για τον πλανόδιο πωλητή), τότε επιλέγουμε ξανά τον δεύτερο γονέα. Τέλος, προσθέτουμε τους 2 γονείς στις αντίστοιχες θέσεις τους στον πίνακα random_samples.

```
# picking randomly the new parents
random_samples = [0]*round(pathsN * 4 / 5)
for j in range(0, round(pathsN * 4 / 5), 2):
    random_samples[j] = random.uniform(0, totalChance)
    random_samples[j], p1idx = parent_picker(random_samples[j],
totalChance, og_population_idx, paths)
    while True:
        random_samples[j + 1] = random.uniform(0, totalChance)
        random_samples[j + 1], p2idx =
parent_picker(random_samples[j+1], totalChance, og_population_idx,
paths)
        # making sure they are not the same person
        if p1idx != p2idx:
            break
```

Πριν την διαδικασία αναπαραγωγής, θα προσθέσουμε το 20% της μερικής ανανέωσης των παλιών μελών στον νέο πληθυσμό (στον πίνακα paths), με τυχαίο τρόπο.

```
# adding some old chromosomes to the new generation (20% of the new
generation)
for idx in range(0, round(pathsN * 1 / 5)):
    paths[idx] = paths[random.randint(0, pathsN - 1)]
```

Διαδικασία αναπαραγωγής:
Για το υπόλοιπο 80%, θα εκπληρώσουμε την παρακάτω διαδικασία όσες φορές όσο το 80% του τωρινού πληθυσμού, με βήμα 2 (στην προκειμένη περίπτωση, 4 φορές). Χρησιμοποιώντας την μέθοδο generate_child() της κλάσης Reproduction (βλέπε πιο κάτω), και τα ζευγάρια των επιλεγμένων γονιών από τον πίνακα random_samples,

παράγουμε 2 παιδιά, ανά 2 γονείς. Αυξάνουμε ανά επανάληψη, τον δείκτη child_counter, για τον πίνακα random_samples, κατά 2.

```
# adding the children (80% of the new population)
for child_idx in range(round(pathsN * 1 / 5), pathsN, 2):
    # adding the children of each "couple"
    paths[child_idx] =
"".join(Reproduction.generate_child(random_samples[child_counter],
random_samples[child_counter+1]))
    paths[child_idx+1] =
"".join(Reproduction.generate_child(random_samples[child_counter+1],
random_samples[child_counter]))
    child_counter += 2
```

Επιλέγουμε την πιθανότητα μετάλλαξης ενός ατόμου ως 10%. Επομένως με πιθανότητα 10%, για κάθε μέλος του νέου πληθυσμού, μεταλλάσσουμε αυτό το μέλος με την μέθοδο mutate (ελεγχόμενη μετάλλαξη).

```
# mutation process (10% chance)
for mut in range(pathsN):
    chance = random.randint(1, 10)
    if chance == 1:
        print("Pre-Mutation: " + str(paths[mut]))
        paths[mut] = "".join(mutate(paths[mut]))
        print("Post-Mutation: " + str(paths[mut]))
```

Τέλος, αυξάνουμε τον αριθμό της γενιάς κατά 1, και ο αλγόριθμος τελειώνει, ελέγχοντας εάν θα επαναληφθεί για άλλη μια γενιά όπως αναφέρθηκα πιο πάνω.

- *reproduction.py*

Περιέχει την κλάση Reproduction, με την μέθοδο generate_child(self, partner):
Είσοδος: self (μεταβλητή string): ο πρώτος γονέας
partner (μεταβλητή string): ο δεύτερος γονέας
Έξοδος: visited (μεταβλητή string): το παιδί των γονιών

Χρησιμοποιούμε αυτή την μέθοδο για την διαδικασία αναπαραγωγής του γενετικού αλγορίθμου. Η μέθοδος αυτή παράγει ένα παιδί, κάνοντας χρήση τους 2 γονείς και το επιστρέφει για να εισαχθεί στον νέο πληθυσμό. Χρησιμοποιούμε διασταύρωση ενός σημείου για την διαδικασία αναπαραγωγής.

Από τον πρώτο γονέα θα πάρουμε την πρώτη και τελευταία πόλη στην διαδρομή που περιέχει (δηλαδή την πόλη Α 2 φορές), καθώς και την δεύτερη και τρίτη πόλη της διαδρομής, και θα τα τοποθετήσουμε στην μεταβλητή visited, στις ίδιες θέσεις που βρίσκονταν στον πρώτο γονέα. Η μεταβλητή visited αντιπροσωπεύει το παιδί των 2 γονέων. Στην προσωρινή μεταβλητή partner_cities, θα προσθέσουμε τις ενδιάμεσες πόλεις της διαδρομής του δεύτερου γονέα (δηλαδή όλες εκτός της Α), με την παρακάτω σειρά: $4^{η}$ πόλη, $5^{η}$ πόλη, $2^{η}$ πόλη, $3^{η}$ πόλη. Έπειτα, για κάθε μια από αυτές τις πόλεις, η μέθοδος θα ελέγξει αν οι πόλεις του partner_cities βρίσκονται ήδη μέσα στο παιδί. Αν όχι, θα προσθέσουμε την αντίστοιχη πόλη του partner_cities στο παιδί visited. Αλλιώς, ο αλγόριθμος συνεχίζει μέχρι να επισκεφτεί όλες τις πόλεις του partner_cities.

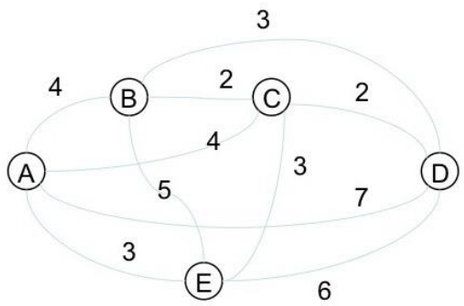
```
def generate_child(self, partner):
    """
    This method is used to generate the child of two members of the
current population and return it
    Args:
        self: The first parent
        partner: The second parent
    """
    # single point crossover (2 cities from 1st parent, 2 cities from
second)

    # all the cities that will be used from the 1st parent
    visited = [self[0], self[1], self[2], self[5]]
```

```
    # all the cities that will be used, if not already, from the 2nd
parent
    partner_cities = [partner[3], partner[4], partner[1], partner[2]]
    # repeat until all cities of the 2nd parent are implemented in
the child trip
    for x in range(len(partner_cities)):
        # if the city is already in the child trip, skip
        if partner_cities[x] in visited:
            continue
        else:
            # the first time a city from the 2nd parent isn't
included, it is inserted as the 4th city in the child
            if len(visited) == 4:
                visited.insert(3, partner_cities[x])
            # the second time a city from the 2nd parent isn't
included, it is inserted as the 5th
            else:
                visited.insert(4, partner_cities[x])
    return visited
```

- *cost_calculator.py*

Περιέχει το λεξικό pathsdict, την κλάση CostCalculator, με την μέθοδο path_cost(self):
Είσοδος: self (μεταβλητή string): το μέλος του τωρινού πληθυσμού που θέλουμε να κοστολογήσουμε
Έξοδος: το κόστος του μονοπατιού του μέλους

Η αναπαράσταση των λύσεων γίνεται με τα σύμβολα string που χρησιμοποιούνται για τις πόλεις στο σχήμα της άσκησης, δηλαδή “A”, “B”, κλπ. Το λεξικό pathsdict, περιέχει το κόστος της κάθε διαδρομής από μία πόλη σε μία άλλη, όπως περιγράφεται στο σχήμα της άσκησης.




```
pathsdict = {
        "AB": 4,
        "AC": 4,
        "AD": 7,
        "AE": 3,
        "BC": 2,
        "BD": 3,
        "BE": 5,
        "CD": 2,
        "CE": 3,
        "DE": 6
    }
```

Κάνουμε χρήση την μέθοδο path_cost(), ως την συνάρτηση καταλληλότητας του γενετικού αλγορίθμου. Δέχεται ένα χρωμόσωμα, δηλαδή μια διαδρομή ενός πλανόδιου πωλητή, μεταξύ των 5 πόλεων του προβλήματος, και το βαθμολογεί με βάση τα κόστη διαδρομών από πόλη σε πόλη που αναγράφονται στο pathsdict. Ύστερα, επιστρέφει το συνολικό κόστος ολόκληρης της διαδρομής

Ξεκινώντας με συνολικό κόστος 0, για κάθε δυάδα πόλεων της διαδρομής, από τις πρώτες 2 πόλεις μέχρι τις 2 τελευταίες, ψάχνουμε στο λεξικό το κόστος ή μάλλον την απόσταση της $1^{ης}$ πόλης με την $2^{η}$. Σε περίπτωση που δεν βρεθεί το αντίστοιχο

μονοπάτι στο λεξικό, αντιστρέφουμε τις αντίστοιχες πόλεις, καθώς η απόσταση μεταξύ 2 πόλεων δεν επηρεάζεται από ποια πόλη ξεκινάει ο πωλητής και σε ποια φτάνει. Έπειτα, προσθέτουμε το αντίστοιχο κόστος στο συνολικό κόστος. Τέλος, επιστρέφουμε την τιμή καταλληλότητας, δηλαδή την πιθανότητα να επιλεχθεί αυτή η διαδρομή, η οποία ισούται με 10/συνολικό κόστος.

```
def path_cost(self):
    """
    This method is used to calculate and return the cost of an entire
trip, from start to end.
    """
    i = 0
    cost: int = 0
    # until the algorithm crosses all of the paths between the cities
on this trip
    while i < len(self) - 2:
        try:
            # search the dictionary pathsdirect, to find the
appropriate cost of this path
            path = self[i] + self[i+1]
            cost += pathsdict[path]
        except:
            # if the path can't be found, reverse the order of the
cities
            path = self[i + 1] + self[i]
            cost += pathsdict[path]
        i += 1

    return 10/cost
```

Παραδείγματα εκτέλεσης

Πρώτο παράδειγμα

```
C:\Users\user\anaconda3\envs\TravelingSalesman\python.exe C:/Users/user/PycharmProjects/TravelingSalesman/main.py
Generation 0: ['AECDBA', 'ADEBCA', 'AECDBA', 'AEDBCA', 'ABCDEA', 'ADBCEA', 'AEBDCA', 'AEBCDA', 'AECBDA', 'ACBDEA']
0.9090909090909091
0.5
0.9090909090909091
0.7142857142857143
0.7142857142857143
0.6666666666666666
0.7692307692307693
0.8333333333333334
0.9090909090909091
0.6666666666666666
7.5917415917415925
Parents:['ADBCEA', 'AEDBCA', 'AEBCDA', 'ABCDEA', 'AEDBCA', 'ADBCEA', 'AECDBA', 'ACBDEA']
Pre-Mutation: ABCDEA
Post-Mutation: ADCBEA
Children:['ADBCEA', 'ACBDEA', 'ADBCEA', 'AEDCBA', 'AEBDCA', 'ADCBEA', 'AEDCBA', 'ADBCEA', 'AECDBA', 'ACBDEA']
```

...μετά από πολλές επαναλήψεις:

```
Generation 221: ['AECDBA', 'ADCEBA', 'AECDBA', 'AECDBA', 'AECDBA', 'AECDBA', 'AECDBA', 'AECDBA', 'AECDBA', 'AECDBA']
0.9090909090909091
0.5882352941176471
0.9090909090909091
0.9090909090909091
0.9090909090909091
0.9090909090909091
0.9090909090909091
0.9090909090909091
0.9090909090909091
0.9090909090909091
8.770053475935828
Parents:['AECDBA', 'AECDBA', 'AECDBA', 'AECDBA', 'AECDBA', 'AECDBA', 'AECDBA', 'AECDBA']
Children:['AECDBA', 'AECDBA', 'AECDBA', 'AECDBA', 'AECDBA', 'AECDBA', 'AECDBA', 'AECDBA', 'AECDBA', 'AECDBA']
```

Δεύτερο παράδειγμα:

```
Generation 0: ['AECBDA', 'ACBEDA', 'ADCBEA', 'ADEBCA', 'ADBECA', 'ABDECA', 'ACDBEA', 'ABECDA', 'ACBEDA', 'ABDECA']
0.9090909090909091
0.5882352941176471
0.625
0.5
0.5555555555555556
0.625
0.7142857142857143
0.7142857142857143
0.5882352941176471
0.625
6.444688481453187
Parents:['ABECDA', 'ADBECA', 'ABECDA', 'AECBDA', 'ABDECA', 'ABDECA', 'ACBEDA', 'ABECDA']
Pre-Mutation: ABDECA
Post-Mutation: ABDCEA
Pre-Mutation: ACBDEA
Post-Mutation: ACEDBA
Pre-Mutation: ABEDCA
Post-Mutation: ABCDEA
Children:['ADEBCA', 'ABDECA', 'ABECDA', 'ADBCEA', 'ABEDCA', 'AECDBA', 'ABDCEA', 'ABDECA', 'ACEDBA', 'ABCDEA']
```

```
Generation 88: ['AECBDA', 'AECBDA', 'AECBDA', 'AECBDA', 'AECBDA', 'ADCBEA', 'AECBDA', 'AECDBA', 'AECBDA', 'AECBDA']
0.9090909090909091
0.9090909090909091
0.9090909090909091
0.9090909090909091
0.9090909090909091
0.625
0.9090909090909091
0.9090909090909091
0.9090909090909091
0.9090909090909091
8.806818181818182
Parents:['AECBDA', 'AECBDA', 'AECBDA', 'AECBDA', 'AECBDA', 'AECBDA', 'AECBDA', 'AECBDA']
Children:['AECBDA', 'AECBDA', 'AECBDA', 'AECBDA', 'AECBDA', 'AECBDA', 'AECBDA', 'AECBDA', 'AECBDA', 'AECBDA']
```