پروژه ژنتیک

قسمت اول : خوشه بندی

1- در ابتدا کتابخانه ها و پکیج های مورد نیاز برای بخش اول این پروژه را ایجاد می کنیم :

```
import pandas as pd
import math
import json
import numpy as np
import random
random.seed(1)
import configparser
NORMALIZATION = True
import matplotlib.pyplot as plt
import seaborn as sns
```

2- حال شروع به ایجاد و تعریف کلاس کروموزوم می کنیم و توابع لازم برای این بخش را ایجاد می کنیم :

```
class Chromosome:
    def __init__(self, genes, length):
        self.genes = genes
        self.length = length
        self.fitness = 0

    def randomGenerateChromosome(self):
        for i in range(0, self.length, +1):
            gen = float('%.2f' % random.uniform(0.0, 1.0))
            self.genes.append(gen)

        return self
```

این بخش از کد یک کلاس را تعریف می‌کند که به عنوان یک کروموزوم در یک الگوریتم ژنتیک عمل می‌کند. در این کلاس، سه ویژگی اصلی تعریف شده‌اند:

1- `genes` : یک لیست از ژن‌ها که کروموزوم را تشکیل می‌دهند.
2- `length` : طول کروموزوم، به عبارت دیگر تعداد ژن‌هایی که کروموزوم دارد.
3- `fitness` : امتیاز یا معیاری که برای سنجش میزان تطابق کروموزوم با مسئله مورد نظر استفاده می‌شود.

سپس یک تابع به نام `randomGenerateChromosome` تعریف شده است که به کلاس اضافه شده است.
این تابع به صورت تصادفی ژن‌های کروموزوم را ایجاد می‌کند. برای هر موقعیت (اندیس) در کروموزوم، یک عدد تصادفی ایجاد شده و به `genes` اضافه می‌شود. سپس کروموزوم جدید ایجاد شده به عنوان خروجی بازگردانده می‌شود.

3- سپس در این بخش کلاس کلاستر را ایجاد کردیم و تابع های مورد نیاز آن را ایجاد می کنیم :

```python
class Cluster:
    def __init__(self, dim, centroid):
        self.dim = dim
        self.centroid = centroid
        self.points = []
        self.distances = []

    def computeS(self):
        n = len(self.points)
        if n == 0:
            return 0
        s = 0
        for x in self.distances:
            s += x
        return float(s / n)
```

این کلاس `Cluster` یک نمایش از یک خوشه در فضای بعدی است. این کلاس دارای ویژگی‌هایی مانند ابعاد خوشه (`dim`)، مرکز خوشه (`centroid`)، لیست نقاطی که به خوشه تعلق دارند (`points`) و لیست فواصل نقاط خوشه تا مرکز آن (`distances`) است.

حالا بیایید به توضیح توابع این کلاس بپردازیم:

1. `__init__` : این تابع مقادیر اولیه خوشه را تنظیم می‌کند. آرگومان‌های ورودی شامل ابعاد خوشه (`dim`) و مرکز خوشه (`centroid`) هستند. همچنین لیست نقاط و فواصل را خالی می‌کند.
2. `computeS` : این تابع فاصله میان نقاط خوشه تا مرکز آن را محاسبه می‌کند و میانگین آن‌ها را برمی‌گرداند. ابتدا تعداد نقاط خوشه را بررسی می‌کند، اگر تعداد نقاط صفر باشد، مقدار صفر را برمی‌گرداند. در غیر این صورت، مقدار فواصل نقاط را جمع می‌کند و میانگین آن‌ها را محاسبه می‌کند و برمی‌گرداند.

سپس در بخش بعدی شروع به پیاده سازی کلاس `clustering` می کنیم که تابع های آن به صورت زیر تعریف می شوند :

این کلاس `Clustering` یک محیط برای اجرای الگوریتم کاهش ابعاد (dimensionality reduction) و خوشه‌بندی (clustering) بر روی داده‌ها فراهم می‌کند. الگوریتمی که این کلاس اجرا می‌کند، مبتنی بر الگوریتم ژنتیک (Genetic Algorithm) است.

1. `__init__` : این تابع مقادیر اولیه برای محیط خوشه‌بندی را تنظیم می‌کند. آرگومان‌های ورودی شامل تعداد نسل‌ها (`generation`)، داده‌های ورودی (`data`) و حداکثر تعداد خوشه‌ها (`kmax`) هستند.
2. `daviesBouldin` : این تابع اندیس دیویس-بولدین را برای خوشه‌های داده شده محاسبه می‌کند.
3. `computeR` و `computeRij` : این دو تابع به ترتیب برای محاسبه مقادیر R و Rij در محاسبات دیویس-بولدین استفاده می‌شوند.
4. `euclidianDistance` : این تابع فاصله اقلیدسی بین دو نقطه را محاسبه می‌کند.
5. `calcDistance` : این تابع برای محاسبه خوشه‌ها و فواصل بین نقاط و خوشه‌ها استفاده می‌شود.
6. `findMin` : این تابع برای یافتن نزدیک‌ترین خوشه به نقطه و اضافه کردن نقطه به آن خوشه استفاده می‌شود.
7. `calcChildFit` : این تابع برای محاسبه وضعیت بچه‌های کروموزوم (individuals) استفاده می‌شود.

8. `calcChromosomesFit` : این تابع برای محاسبه وضعیت تمامی کروموزوم‌ها در یک نسل مشخص استفاده می‌شود.
9. `printIBest` : این تابع برای چاپ وضعیت بهترین کروموزوم در یک نسل استفاده می‌شود، از جمله محاسبه دقت و چاپ مرکزهای خوشه‌ها.
10. `output_result` : این تابع نتایج خوشه‌بندی را به فایل‌ها ذخیره می‌کند، شامل مراکز خوشه‌ها و نقاط مرکزی هر خوشه در یک فایل JSON و نتایج خوشه‌بندی در یک فایل CSV است.

کلاس `Generation` مربوط به مدلی از نسل در یک الگوریتم ژنتیک است. در الگوریتم ژنتیک، یک نسل شامل یک مجموعه از کروموزوم‌ها (individuals) است.

```
class Generation:
    def __init__(self, numberOfIndividual, generationCount):
        self.numberOfIndividual = numberOfIndividual
        self.chromosomes = []
        self.generationCount = generationCount

    def sortChromosomes(self):
        self.chromosomes = sorted(
            self.chromosomes, reverse=True, key=lambda elem: elem.fitness)
        return self.chromosomes

    def randomGenerateChromosomes(self, lengthOfChromosome):
        for i in range(0, self.numberOfIndividual):
            chromosome = Chromosome([], lengthOfChromosome)
            chromosome.randomGenerateChromosome()
            self.chromosomes.append(chromosome)
```

حالا بیایید به توضیح توابع این کلاس بپردازیم:

1. `__init__` : این تابع مقادیر اولیه برای یک نسل را تنظیم می‌کند. آرگومان‌های ورودی شامل تعداد کروموزوم‌ها (`numberOfIndividual`)، یک لیست از کروموزوم‌ها (`chromosomes`) و تعداد نسل (`generationCount`) هستند.
2. `sortChromosomes` : این تابع برای مرتب‌سازی کروموزوم‌ها بر اساس وضعیت فیتنس آن‌ها استفاده می‌شود. به عبارت دیگر، کروموزوم‌ها بر اساس امتیاز فیتنسی خود مرتب می‌شوند.
3. `randomGenerateChromosomes` : این تابع برای تولید تصادفی کروموزوم‌ها به تعداد مشخص شده استفاده می‌شود. ابتدا یک شیء از کلاس `Chromosome` ایجاد می‌شود و سپس توابع مربوط به تولید تصادفی کروموزوم فراخوانده می‌شوند تا یک کروموزوم تولید شود. سپس کروموزوم به لیست کروموزوم‌های نسل اضافه می‌شود.

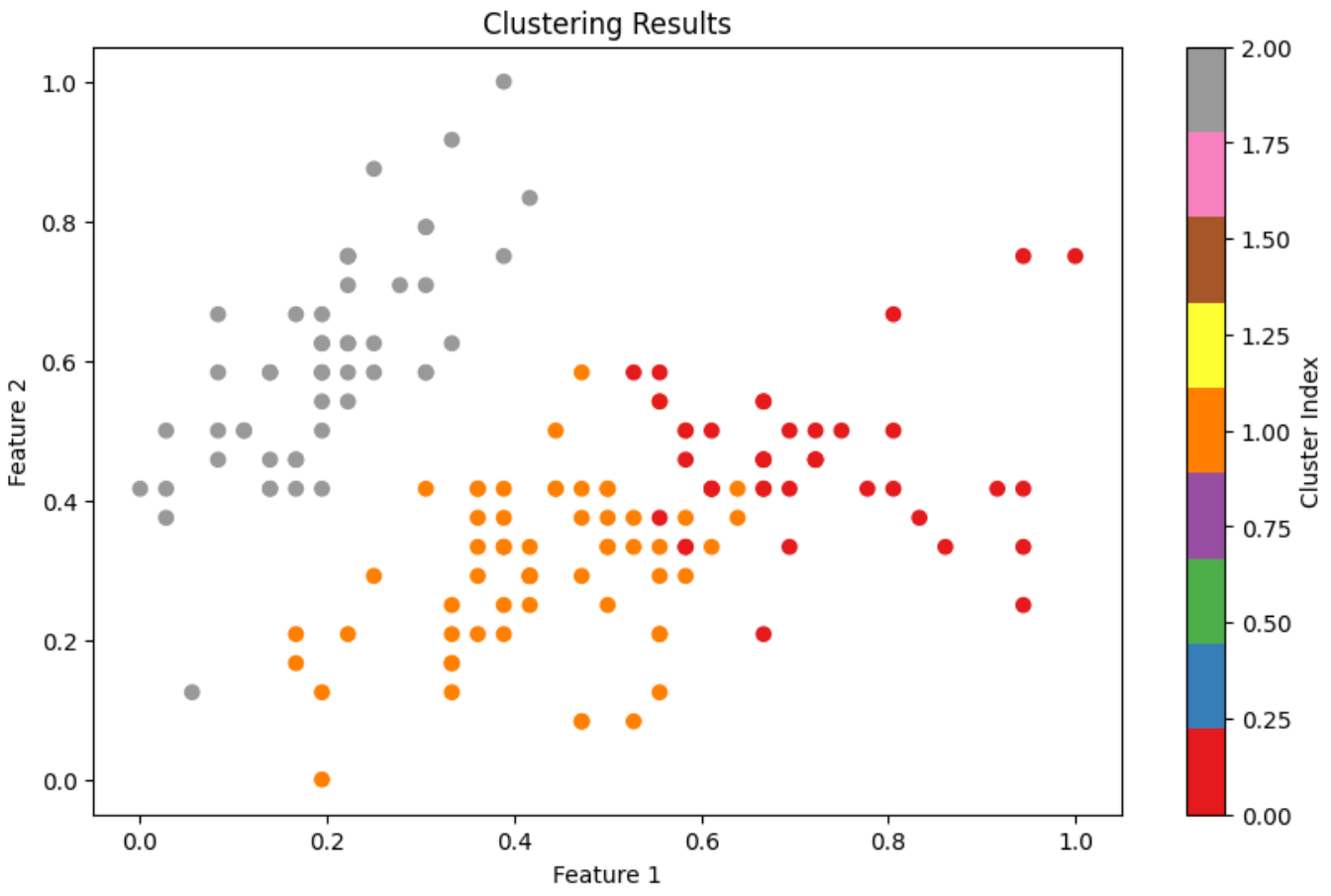
حالا در این بخش به تعریف کلاس `Genetic` می کنیم که مهمترین کلاس کد ما می باشد.

حالا بیایید به توضیح توابع این کلاس بپردازیم:

1. `__init__` : این تابع مقادیر اولیه برای اجرای الگوریتم ژنتیک را تنظیم می‌کند. آرگومان‌های ورودی شامل تعداد کروموزوم‌ها (`numberOfIndividual`)، احتمال انتخاب یک کروموزوم برای بقایش در نسل بعدی (`Ps`)، احتمال جهش (`Pm`)، احتمال چندگانه شدن (`Pc`)، بودجه (`budget`)، داده‌ها (`data`)، تعداد نسل (`generationCount`) و حداکثر تعداد خوشه‌ها (`kmax`) هستند.

2. `geneticProcess` : این تابع مراحل اصلی الگوریتم ژنتیک را اجرا می‌کند، شامل انتخاب، جهش و چندگانه شدن کروموزوم‌ها است.
3. `selection` : این تابع برای انتخاب کروموزوم‌ها بر اساس روش انتخاب رتبه‌ای (Ranking Selection) استفاده می‌شود.
4. `crossover` : این تابع برای انجام عملیات چندگانه شدن (Crossover) روی کروموزوم‌ها استفاده می‌شود.
5. `doCrossover` : این تابع برای انجام واقعی چندگانه شدن بین دو کروموزوم استفاده می‌شود و نتایج آن را با محاسبه فیتنس به دست می‌آورد.
6. `mutation` : این تابع برای انجام عملیات جهش روی کروموزوم‌ها استفاده می‌شود.
7. `doMutation` : این تابع برای انجام واقعی عملیات جهش بر روی یک کروموزوم استفاده می‌شود و نتایج آن را با محاسبه فیتنس به دست می‌آورد.
8. `readVars` : این تابع برای خواندن پارامترهای مورد نیاز از یک فایل پیکربندی استفاده می‌شود.
9. `minmax` : این تابع برای اعمال عملیات نرمال‌سازی (MinMax Normalization) بر روی داده‌ها استفاده می‌شود.
10. `main` : این قسمت برنامه، اجرای اصلی الگوریتم ژنتیک و خوشه‌بندی را بر روی داده‌ها انجام می‌دهد، شامل تولید نسل اولیه، محاسبه فیتنس، اجرای الگوریتم ژنتیک و ذخیره‌ی نتایج است.

خروجی برنامه برای بخش اول :



قسمت دوم : گراف

```python
def initialize_population(nodes,pop_size):
    max_nod_num = max(nodes)
    population = []
    for i in range(pop_size):
        chromosome = []
        # to create a fully connected path
        while len(chromosome)!=len(nodes):
            rand_node = np.random.randint(max_nod_num+1)
            # to prevent repeted additions of nodes in the same chrromosome
            if rand_node not in chromosome:
                chromosome.append(rand_node)
        population.append(chromosome)
    return population
```

این تابع `initialize_population` یک جمعیت اولیه برای الگوریتم ژنتیک را ایجاد می‌کند که شامل چندین کروموزوم است. هر کروموزوم در این جمعیت نشان‌دهنده یک مسیر (path) در گراف است.

وارداتهای مورد نیاز را به متغیرها می‌دهد و سپس برای هر کروموزوم در جمعیت، یک مسیر را به صورت تصادفی ایجاد می‌کند. این مسیر به صورت یک لیست از گره‌ها (nodes) نشان داده می‌شود.

برای ایجاد هر کروموزوم، ابتدا یک لیست خالی به نام `chromosome` ایجاد می‌شود. سپس در یک حلقه while، تا زمانی که تعداد گره‌های مسیر برابر با تعداد گره‌های موجود در گراف نشود، عملیات زیر انجام می‌شود:

1. یک شماره گره تصادفی (بین گره‌های موجود در گراف) انتخاب می‌شود و در `rand_node` ذخیره می‌شود.
2. اگر این گره قبلاً به `chromosome` اضافه نشده باشد، آن را به انتهای `chromosome` اضافه می‌کند.

این عملیات تا زمانی ادامه می‌یابد که تمامی گره‌ها به `chromosome` اضافه شوند، سپس `chromosome` به عنوان یک کروموزوم جدید به `population` اضافه می‌شود.

در نهایت، `population` حاوی تعداد مشخص شده از کروموزوم‌ها (جمعیت) با مسیرهای تصادفی در گراف است و به عنوان خروجی تابع برگردانده می‌شود.

سپس در بخش بعدی شروع به پیاده سازی تابع cost می کنیم :

```
def cost(graph_edges,chromosome):
        total_cost=0
        i=1
        while i<len(chromosome):
            for temp_edge in graph_edges:
                if chromosome[i-1]==temp_edge[0] and
                chromosome[i]==temp_edge[1]:
                    total_cost=total_cost+temp_edge[2]
            i=i+1
        for temp_edge in graph_edges:
                if chromosome[0]==temp_edge[0] and
                chromosome[len(chromosome) - 1]==temp_edge[1]:
                    total_cost=total_cost+temp_edge[2]
        return total_cost
```

این تابع `cost` مجموع هزینه مسیر مشخص شده توسط کروموزوم را در یک گراف مشخص محاسبه می‌کند.

آرگومان‌های ورودی این تابع عبارتند از:

- `graph_edges` : لیستی از یال‌ها (توابع محتوی نودهایی است که توسط یک یال به هم متصل شده‌اند و وزن یال) که هر یال از نود مبدا، نود مقصد و هزینه یال را نشان می‌دهد.
- `chromosome` : یک مسیر در گراف که توسط یک کروموزوم مشخص شده است. این مسیر به صورت لیستی از نودها مشخص می‌شود.

در این تابع، ابتدا متغیر `total_cost` را برابر صفر قرار داده و سپس در یک حلقه، به طول کروموزوم پیش می‌رویم. در هر مرحله، برای هر یال در `graph_edges` ، بررسی می‌کنیم که آیا یال متناظر با گره فعلی و گره بعدی که در کروموزوم ذکر شده‌اند، موجود است یا خیر. اگر موجود باشد، هزینه آن یال به `total_cost` اضافه می‌شود.

سپس، بررسی می‌شود که آیا یالی وجود دارد که اولین و آخرین گره‌های کروموزوم را به هم وصل می‌کند یا خیر. اگر وجود داشت، هزینه آن نیز به `total_cost` اضافه می‌شود.

در نهایت، مقدار `total_cost` که هزینه کل مسیر است، به عنوان خروجی تابع برگردانده می‌شود.

سپس شروع به تعریف تابع Select_best می کنیم :

```
def select_best(parent_gen,graph_edges,elite_size):
    costs = []
    selected_parent = []
    pop_fitness = []
    for i in range(len(parent_gen)):
        costs.append(cost(graph_edges,parent_gen[i]))
        pop_fitness.append((costs[i],parent_gen[i]))
    #sort according to path_costs
    pop_fitness.sort(key = lambda x: x[0])
    # select only top elite_size fittest chromosomes in the population
    for i in range(elite_size):
        selected_parent.append(pop_fitness[i][1])
    return selected_parent,pop_fitness[0][0],selected_parent[0]
```

این تابع `select_best` از بین کروموزوم‌های موجود در جمعیت والد (پدر)، بهترین‌ها را انتخاب می‌کند و به همراه هزینه مسیر بهترین کروموزوم و خود بهترین کروموزوم را برمی‌گرداند.

آرگومان‌های ورودی این تابع عبارتند از:

- `parent_gen` : لیستی از کروموزوم‌های موجود در جمعیت والد (پدر).
- `graph_edges` : لیستی از یال‌ها (توابع محتوی نودهایی است که توسط یک یال به هم متصل شده‌اند و وزن یال) که هر یال از نود مبدا، نود مقصد و هزینه یال را نشان می‌دهد.
- `elite_size` : تعداد کروموزوم‌های برتر (پدرهای الیت) که انتخاب می‌شوند.

در ابتدا، یک لیست به نام `costs` برای ذخیره هزینه‌های مسیرهای مربوط به هر کروموزوم و یک لیست به نام `pop_fitness` برای ذخیره جفت هزینه و کروموزوم مربوطه ایجاد می‌شود.

سپس، برای هر کروموزوم موجود در `parent_gen` ، هزینه مسیر متناظر با آن با استفاده از تابع `cost` محاسبه شده و به `costs` اضافه می‌شود. همچنین، هر جفت هزینه و کروموزوم مربوطه در `pop_fitness` ذخیره می‌شود.

سپس، `pop_fitness` بر اساس هزینه‌های مسیر مرتب می‌شود.

در ادامه، تنها اولین `elite_size` کروموزوم از `pop_fitness` انتخاب شده و به `selected_parent` اضافه می‌شوند.

در نهایت، لیست `selected_parent` که شامل بهترین کروموزوم‌ها است، همچنین هزینه مسیر بهترین کروموزوم و خود بهترین کروموزوم (اولین عضو در `selected_parent`) به عنوان خروجی تابع برگردانده می‌شوند.

دو تابع، `breed` و `breedPopulation` ، برای تولید فرزندان جدید از والدین (پدر و مادر) در فرآیند تولید نسل بعدی در الگوریتم ژنتیک استفاده می‌شوند.

```python
def breed(parent1, parent2):
    child = []
    childP1 = []
    childP2 = []

    geneA = int(random.random() * len(parent1))
    geneB = int(random.random() * len(parent1))
    if geneA < geneB :
        startGene, endGene = geneA, geneB
    else :
        endGene, startGene = geneA, geneB

    for i in range(startGene, endGene):
        childP1.append(parent1[i])

    childP2 = [item for item in parent2 if item not in childP1]

    child = childP1 + childP2
    return child
```

```python
def breedPopulation(parents, pop_size):
    children = []
    temp = np.array(parents)
    n_parents = temp.shape[0]
    #create new population of size pop_size from previous population
    for i in range(pop_size):
        #choose random parents
        random_dad = parents[np.random.randint(low =0,high = n_parents - 1)]
        random_mom = parents[np.random.randint(low =0,high = n_parents - 1)]
        #create child using random parents
        children.append(breed(random_dad,random_mom))
    return children
```

تابع `breed` :

این تابع دو کروموزوم را به عنوان ورودی دریافت می‌کند و فرزندی جدید را با استفاده از آن‌ها تولید می‌کند. روش تولید فرزند از والدین به این صورت است:

1. ابتدا دو نقطه تصادفی را در نواحی مختلف از والدین انتخاب می‌کند. این دو نقطه با نام `geneA` و `geneB` نمایانگر این نقاط هستند.
2. سپس بررسی می‌شود که کدام نقطه ابتدا و کدام نقطه انتها است. بر اساس این بررسی، نواحی مختلف از والدین برای استفاده در فرزند تعیین می‌شوند.
3. بخشی از یک والد به صورت یکپارچه به فرزند منتقل می‌شود.
4. نواحی باقی‌مانده از والد دیگر به فرزند اضافه می‌شوند.
5. فرزند حاصل به عنوان خروجی تابع برگردانده می‌شود.

تابع `breedPopulation` :

این تابع یک جمعیت اولیه از والدین را دریافت می‌کند و از آن‌ها فرزندان جدید را تولید می‌کند. تعداد فرزندان تولید شده برابر با اندازه مورد نظر جمعیت جدید است. این تابع به ازای هر فرزند ابتدا دو والد را به صورت تصادفی از جمعیت اولیه انتخاب می‌کند، سپس با استفاده از تابع `breed` فرزند جدید را تولید می‌کند. فرزندان تولید شده به عنوان خروجی تابع برگردانده می‌شوند.

```python
def mutate(parent, n_mutations):
    # we cannot randomly change a node from chromosome to another node
    # as this will create repeated nodes
    # we define mutation as mutation of edges in the path i.e swapping of
    #nodes in the chromosome
    temp_parent = np.array(parent)
    size1 = temp_parent.shape[0]
    max_nod_num = max(parent)
    for i in range(n_mutations):
        # choose random indices to swap nodes in a chromosome
        rand1 = np.random.randint(0,size1)
        rand2 = np.random.randint(0,size1)
        # if rand1 and rand2 are same, then chromosome won't be mutated
        # so change rand2
        if rand1 == rand2:
            rand2=(rand2+1)%size1
        parent[rand1],parent[rand2]=parent[rand2],parent[rand1]
    return parent


def mutatePopulation(population, n_mutations):
    mutatedPop = []
    #mutate population
    for ind in range(0, len(population)):
        mutatedInd = mutate(population[ind], n_mutations)
        mutatedPop.append(mutatedInd)
    return mutatedPop
```

تابع `mutate` :

1. ورودی‌های این تابع شامل `parent` (کروموزوم اصلی که قرار است متحول شود) و `n_mutations` (تعداد جفت نقاطی که برای جابجایی در کروموزوم انتخاب می‌شود) است.
2. این تابع به طور تصادفی چندین بار یک جفت نقطه از کروموزوم را انتخاب می‌کند و آن‌ها را جابجا می‌کند. این جابجایی‌ها به عنوان جهش در فرآیند انتخاب طبیعی شناخته می‌شوند.
3. اگر نقطه‌های انتخابی برابر باشند، هیچ تغییری در کروموزوم ایجاد نمی‌شود.
4. نتیجهٔ نهایی این تابع یک کروموزوم متحول شده است که از جهش‌های انجام شده ناشی می‌شود.

تابع `mutatePopulation` :

1. ورودی این تابع شامل `population` (جمعیت کروموزوم‌ها که در هر نسل وجود دارد) و `n_mutations` (تعداد جفت نقاطی که برای هر کروموزوم برای جابجایی انتخاب می‌شود) است.
2. این تابع به ازای هر کروموزوم در جمعیت، تابع `mutate` را فراخوانی می‌کند تا کروموزوم متحول شود.
3. نتیجهٔ نهایی این تابع یک جمعیت جدید است که شامل کروموزوم‌هایی است که از جمعیت اولیه با انجام جهش‌ها به دست آمده‌اند.

سپس در این بخش شروع به پیاده سازی کلاس `Graph` می کنیم که به صورت زیر می باشد :

در ادامه، توضیح مختصری از هر یک از متدهای این کلاس آمده است:

1. `__init__(self, vertices)` : سازنده کلاس است که یک شیء گراف را ایجاد می‌کند. ورودی آن شامل لیستی از رئوس گراف می‌باشد.
2. `addEdge(self,u,v,w)` : این متد یک یال جدید به گراف اضافه می‌کند. ورودی‌های آن شامل رأس مبدأ (`u`)، رأس مقصد (`v`) و وزن یال (`w`) می‌باشند.
3. `get_cost(self,visited_nodes)` : این متد هزینه مسیری که توسط لیستی از رئوس (`visited_nodes`) طی شده است را محاسبه می‌کند.
4. `disconnected(self,initial_node)` : این متد بررسی می‌کند که گراف دارای اتصال کامل است یا خیر.
5. `gen_algo(self,source,generations)` : این متد الگوریتم ژنتیک را بر روی گراف اجرا می‌کند تا بهینه‌ترین مسیر را بین رأس مبدأ و تمام رئوس دیگر گراف بیابد. این الگوریتم شامل مراحل مختلفی مانند ایجاد جمعیت اولیه، انتخاب والدین برتر، تولید نسل جدید (تولید فرزندان)، و جهش‌های ژنتیک می‌شود.

به طور خلاصه، این کلاس یک ساختار داده است که یک گراف را نمایش می‌دهد و امکاناتی را برای محاسبه مسایل مختلفی مانند هزینه مسیر بهینه و اجرای الگوریتم‌های بهینه‌سازی ارائه می‌دهد.

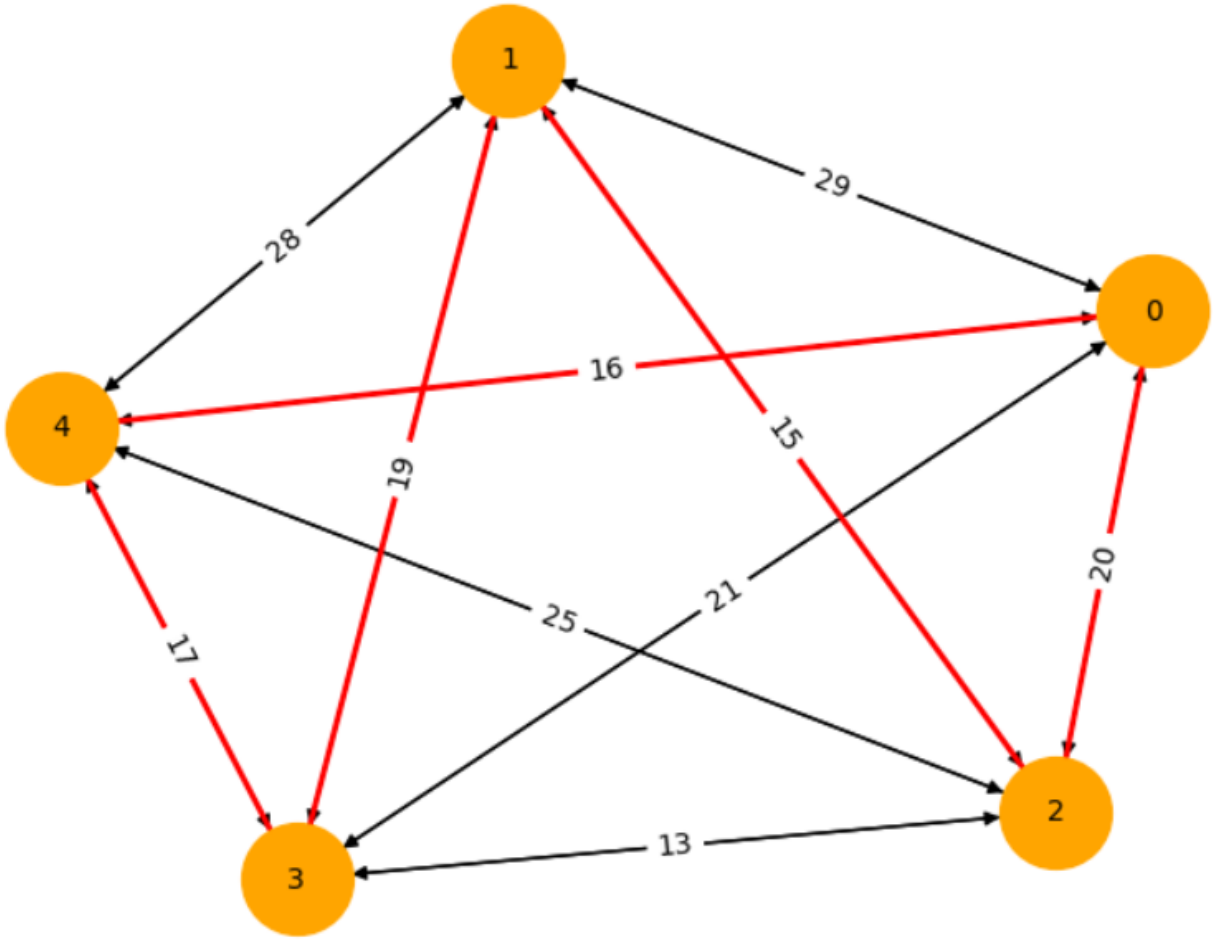
سپس با توجه به گرافی که در سوال داده شده است مسیر مورد نظر و هزینه رسیدن به آن را حساب می کنیم (منظور هزینه بهینه رسیدن به هدف می باشد)

و در نهایت خروجی سوال حل شده را نمایش می دهیم :

```
==========================================================================================================
Generation number : 96 / 100
Best route for generation 96 : [3, 4, 0, 2, 1]
Best cost for generation 96 : 87
==========================================================================================================
Generation number : 97 / 100
Best route for generation 97 : [0, 4, 3, 1, 2]
Best cost for generation 97 : 87
==========================================================================================================
Generation number : 98 / 100
Best route for generation 98 : [2, 0, 4, 3, 1]
Best cost for generation 98 : 87
==========================================================================================================
Generation number : 99 / 100
Best route for generation 99 : [3, 4, 0, 2, 1]
Best cost for generation 99 : 87
==========================================================================================================
Generation number : 100 / 100
Best route for generation 100 : [2, 0, 4, 3, 1]
Best cost for generation 100 : 87
==========================================================================================================
================Path found================
final path:
0 -> 4
4 -> 3
3 -> 1
1 -> 2
2 -> 0
total_cost 87
```

و در نهایت گراف مورد نظر را با استفاده از هزینه کمینه آن چاپ می کنیم :

Shortest path using TSP algorithm



The most optimal route: 87