```python
# -*- coding: utf-8 -*-
"""
Created on Tue Sep 10 17:28:33 2018

@author: CarlosF


Genetic algorithms and the Traveling Salesman Problem
    https://web.cs.elte.hu/blobs/diplomamunkak/bsc_alkmat/2017/keresztury_b

Evolutionary algorithm to traveling salesman problems
    https://www.sciencedirect.com/science/article/pii/S089812211101073X

Genetic algorithms for the traveling salesman problem
    https://iccl.inf.tu-dresden.de/w/images/b/b7/GA_for_TSP.pdf

"""

import numpy as np
import random, operator
import pandas as pd
import matplotlib.pyplot as plt


class City:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, city):
        xDis = abs(self.x - city.x)
        yDis = abs(self.y - city.y)
        distance = np.sqrt((xDis ** 2) + (yDis ** 2))
        return distance

    def __repr__(self):
        return "(" + str(self.x) + "," + str(self.y) + ")"

    def to_dict(self):

        return {
            'x': self.x,
            'y': self.y,
        }


class Fitness:
```

```python
    def __init__(self, route):
        self.route = route
        self.distance = 0
        self.fitness= 0.0

    def routeDistance(self):
        if self.distance ==0:
            pathDistance = 0
            for i in range(0, len(self.route)):
                fromCity = self.route[i]
                toCity = None
                if i + 1 < len(self.route):
                    toCity = self.route[i + 1]
                else:
                    toCity = self.route[0]
                pathDistance += fromCity.distance(toCity)
            self.distance = pathDistance
        return self.distance

    def fit(self):
        if self.fitness == 0:
            self.fitness = 1 / float(self.routeDistance())
        return self.fitness



def createChromosome(cityList):
    random.seed()
    chromosome = random.sample(cityList, len(cityList))
    return chromosome


def initialPopulationFromCityList(popSize,cityList):
    population = []
    for i in range(0, popSize):
        population.append(createChromosome(cityList))
    return population


def calcAdaptation(population):
    fitnessResults = {}
    for i in range(0,len(population)):
        fitnessResults[i] = Fitness(population[i]).fit()
    return sorted(list(fitnessResults.items()), key = operator.itemgetter(
```

```
def selection(popRanked, eliteSize):
    selectionResults = []
    df = pd.DataFrame(np.array(popRanked), columns=["Index","Fitness"])
    df['cum_sum'] = df.Fitness.cumsum()
    df['cum_perc'] = 100*df.cum_sum/df.Fitness.sum()

    for i in range(0, eliteSize):
        selectionResults.append(popRanked[i][0])
    for i in range(0, len(popRanked) - eliteSize):
        pick = 100*random.random()
        for i in range(0, len(popRanked)):
            if pick <= df.iat[i,3]:
                selectionResults.append(popRanked[i][0])
                break
    return selectionResults

def matingPool(population, selectionResults):
    matingpool = []
    for i in range(0, len(selectionResults)):
        index = selectionResults[i]
        matingpool.append(population[index])
    return matingpool

def ordered_crossover(parent1, parent2):
    child = []
    childP1 = []
    childP2 = []

    geneA = int(random.random() * len(parent1))
    geneB = int(random.random() * len(parent1))

    startGene = min(geneA, geneB)
    endGene = max(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

def crossoverPopulation(matingpool, eliteSize):
    children = []
    length = len(matingpool) - eliteSize
```

```
    pool = random.sample(matingpool, len(matingpool))

    for i in range(0,eliteSize):
        children.append(matingpool[i])

    for i in range(0, length):
        child = ordered_crossover(pool[i], pool[len(matingpool)-i-1])
        children.append(child)
    return children

def mutate(individual, mutationRate):
    for swapped in range(len(individual)):
        if(random.random() < mutationRate):
            swapWith = int(random.random() * len(individual))

            city1 = individual[swapped]
            city2 = individual[swapWith]

            individual[swapped] = city2
            individual[swapWith] = city1
    return individual

def mutatePopulation(population, mutationRate):
    mutatedPop = []

    for ind in range(0, len(population)):
        mutatedInd = mutate(population[ind], mutationRate)
        mutatedPop.append(mutatedInd)
    return mutatedPop

def nextGeneration(currentGen, eliteSize, mutationRate):
    popRanked = calcAdaptation(currentGen)
    selectionResults = selection(popRanked, eliteSize)
    matingpool = matingPool(currentGen, selectionResults)
    children = crossoverPopulation(matingpool, eliteSize)
    nextGeneration = mutatePopulation(children, mutationRate)
    return nextGeneration


def geneticAlgorithm(population, popSize, eliteSize, mutationRate, generat
    pop = initialPopulation(popSize, population)
    print(("Initial distance: " + str(1 / rankRoutes(pop)[0][1])))

    for i in range(0, generations):
        pop = nextGeneration(pop, eliteSize, mutationRate)

    print(("Final distance: " + str(1 / rankRoutes(pop)[0][1])))
```

```
    bestRouteIndex = rankRoutes(pop)[0][0]
    bestRoute = pop[bestRouteIndex]
    return bestRoute



def geneticAlgorithmPlot(population, popSize, eliteSize, mutationRate, gene
    iPopulation = initialPopulation(popSize, population)
    progress = []
    progress.append(1 / rankRoutes(iPopulation)[0][1])

    for i in range(0, generations):
        iPopulation = nextGeneration(iPopulation, eliteSize, mutationRate)
        progress.append(1 / rankRoutes(iPopulation)[0][1])

    plt.plot(progress)
    plt.ylabel('Distance')
    plt.xlabel('Generation')
    plt.show()



import networkx as nx
import itertools

cityObjList = []
cityIdxlist = []

for i in range(0,25):
    cityObjList.append(City(x=int(random.random() * 200), y=int(random.ranc
    cityIdxlist.append(i);

idxConection    = list(itertools.permutations(cityIdxlist,2))

for idx,p in enumerate(idxConection):
    idxConection[idx] = idxConection[idx] + (cityObjList[idxConection[idx]]
labels = ['node1','node2','distance']
df_edgelist = pd.DataFrame.from_records(idxConection,columns=labels)


df_nodelist = pd.DataFrame.from_records([city.to_dict() for city in cityObj
idx = 0
df_nodelist.insert(loc=idx, column='node', value=cityIdxlist)



g = nx.Graph()
```

```
for i, elrow in df_edgelist.iterrows():
    g.add_edge(int(elrow[0]), int(elrow[1]),weight=elrow[2])

for i, nlrow in df_nodelist.iterrows():
    g.node[nlrow['node']]['X'] = nlrow[1]
    g.node[nlrow['node']]['Y'] = nlrow[2]
    g.node[nlrow['node']]['node_color'] = 'b'
    g.node[nlrow['node']]['alpha'] = 0.8



g.edges(data=True)
g.nodes(data=True)

print('# of edges: {}'.format(g.number_of_edges()))
print('# of nodes: {}'.format(g.number_of_nodes()))

node_positions = {node[0]: (node[1]['X'], -node[1]['Y']) for node in g.node


plt.figure(figsize=(8, 6))
nx.draw(g, pos=node_positions, node_size=700, node_color='red')
nx.draw_networkx_labels(g, node_positions, font_size=12, font_family='sans
plt.show()

import statistics

popSize=150
eliteSize=20
mutationRate=0.01
generations=400
num_samples = 5


listaProgresso = np.zeros((num_samples,generations+1))

for n in range(0, num_samples):
    iPopulation = initialPopulationFromCityList(popSize,cityObjList)

    bestPaths = []
    progress = []
    dev = []

    progress.append(1 / calcAdaptation(iPopulation)[0][1])
    temp = calcAdaptation(iPopulation)
    dev.append(statistics.stdev([1/upla[1] for upla in temp]))
    bestPaths.append(iPopulation[0])
```

```python
    for i in range(0, generations):
        iPopulation = nextGeneration(iPopulation, eliteSize, mutationRate)
        progress.append(1 / calcAdaptation(iPopulation)[0][1])
        temp = calcAdaptation(iPopulation)
        dev.append(statistics.stdev([upla[1] for upla in temp]))
        bestPaths.append(iPopulation[0])

    listaProgresso[n,:] = progress


# Show min dist by generation

plt.figure(figsize=(8, 6))
for n in range(0, num_samples):
    plt.plot(np.arange(0,generations+1),listaProgresso[n],'-')
plt.ylabel('Tamanho da menor trajetória')
plt.xlabel('Geração')
plt.show()

std = statistics.stdev(listaProgresso[:,-1])
statistics.mean(listaProgresso[:,-1])

plt.figure()
plt.hist(df_edgelist['distance'],bins=20)
plt.axvline(x=std,color='k', linestyle='--')
plt.ylabel('Quantidade de Arestas')
plt.xlabel('distância')
plt.show()

# Show stdev of population by generation

plt.figure(figsize=(8, 6))
plt.plot(dev[2:])
plt.ylabel('Desvio Padrão')
plt.xlabel('Geração')
plt.show()

# Plot Network with best Path

dicNode = dict(zip(cityObjList,cityIdxlist))

bestPath = bestPaths[-1]
bestNodes = [dicNode[n] for n in bestPath]
links = []
for i in range(0,len(bestNodes)-1):
    links.append( (bestNodes[i],bestNodes[i+1],) )
```

```
plt.figure(figsize=(8, 6))
nx.draw(g, pos=node_positions, node_size=700)
nx.draw_networkx_edges(g, node_positions, edgelist=links,
                       width=3, edge_color='b', style='dashed')
nx.draw_networkx_labels(g, node_positions, font_size=12, font_family='sans
plt.show()
```