**TASK 1:**

**Write Python programs to implement stimulated annealing algorithm that solve the salesman travelling problem.**

**CODE:**

**TSP:**

import math

import random

import numpy as np

def stimulated\_annealing(u,T,cool):

        count =0

        start = u \* math.sin(1/(0.01+ math.pow(u,2))) + math.pow(u,3) \* math.sin(1/(0.01+ math.pow(u,4)))

        while T > 0.05:

            for i in range(100):

                direction\_u = random.uniform(-1,1)

                s1 = direction\_u \* math.sin(1/(0.01+ math.pow(direction\_u,2))) + math.pow(direction\_u,3) \* math.sin(1/(0.01+ math.pow(direction\_u,4)))

                if s1 > start:

                    start = s1

                if s1 == start:

                    start = s1

                elif random.random() < math.exp((s1 - start) / T):

                    start = s1

            T\*=cool

            count+=1

            print(str(count) +") z : %s" %start, "u : %s" %direction\_u, "Temperature : %s" %T)

stimulated\_annealing(5,150,0.7)

**Nodes:**

import random

import numpy as np

class NodeGenerator:

    def \_\_init\_\_(self, width, height, nodesNumber):

        self.width = width

        self.height = height

        self.nodesNumber = nodesNumber

    def generate(self):

        xs = np.random.randint(self.width, size=self.nodesNumber)

        ys = np.random.randint(self.height, size=self.nodesNumber)

        return np.column\_stack((xs, ys))

**Simulated Anealing:**

import math

import random

import matplotlib.pyplot as plt

import tsp\_utils

import animated\_visualizer

class SimulatedAnnealing:

    def weight(self, sol):'

        return sum([self.dist\_matrix[i, j] for i, j in zip(sol, sol[1:] + [sol[0]])])

    def acceptance\_probability(self, candidate\_weight):

        return math.exp(-abs(candidate\_weight - self.curr\_weight) / self.temp)

    def accept(self, candidate):

        candidate\_weight = self.weight(candidate)

        if candidate\_weight < self.curr\_weight:

            self.curr\_weight = candidate\_weight

            self.curr\_solution = candidate

            if candidate\_weight < self.min\_weight:

                self.min\_weight = candidate\_weight

                self.best\_solution = candidate

        else:

            if random.random() < self.acceptance\_probability(candidate\_weight):

                self.curr\_weight = candidate\_weight

                self.curr\_solution = candidate

    def anneal(self):

        while self.temp >= self.stopping\_temp and self.iteration < self.stopping\_iter:

            candidate = list(self.curr\_solution)

            l = random.randint(2, self.sample\_size - 1)

            i = random.randint(0, self.sample\_size - l)

            candidate[i: (i + l)] = reversed(candidate[i: (i + l)])

            self.accept(candidate)

            self.temp \*= self.alpha

            self.iteration += 1

            self.weight\_list.append(self.curr\_weight)

            self.solution\_history.append(self.curr\_solution)

        print('Minimum weight: ', self.min\_weight)

        print('Improvement: ',

              round((self.initial\_weight - self.min\_weight) / (self.initial\_weight), 4) \* 100, '%')

    def animateSolutions(self):

        animated\_visualizer.animateTSP(self.solution\_history, self.coords)

    def plotLearning(self):

        plt.plot([i for i in range(len(self.weight\_list))], self.weight\_list)

        line\_init = plt.axhline(y=self.initial\_weight, color='r', linestyle='--')

        line\_min = plt.axhline(y=self.min\_weight, color='g', linestyle='--')

        plt.legend([line\_init, line\_min], ['Initial weight', 'Optimized weight'])

        plt.ylabel('Weight')

        plt.xlabel('Iteration')

        plt.show()

**OUTPUT:**

