AHMAD RAZA

19F-0151

User

[Company name]  [Company address]

**ASSIGNMENT 2:**

**TASK:**

import copy  
import math  
import sys  
import time  
import random  
  
  
def round\_figures(x, n):  
 return round(x, int(n - math.ceil(math.log10(abs(x)))))  
  
  
def time\_string(seconds):  
 s = int(round(seconds)) # round to nearest second  
 h, s = divmod(s, 3600) # get hours and remainder  
 m, s = divmod(s, 60) # split remainder into minutes and seconds  
 return '%4i:%02i:%02i' % (h, m, s)  
  
  
class Annealer(object):  
  
  
 Tmax = 25000.0  
 Tmin = 2.5  
 steps = 50000  
 updates = 100  
 copy\_strategy = 'deepcopy'  
  
  
def \_init\_(self, initial\_state):  
 self.initial\_state = initial\_state  
 self.state = self.copy\_state(initial\_state)  
  
  
def set\_schedule(self, schedule):  
  
 self.Tmax = schedule['tmax']  
 self.Tmin = schedule['tmin']  
 self.steps = int(schedule['steps'])  
  
  
def copy\_state(self, state):  
  
 if self.copy\_strategy == 'deepcopy':  
 return copy.deepcopy(state)  
 elif self.copy\_strategy == 'slice':  
 return state[:]  
 elif self.copy\_strategy == 'method':  
 return state.copy()  
  
  
def update(self, step, T, E, acceptance, improvement):  
 elapsed = time.time() - self.start  
 if step == 0:  
 print(' Temperature Energy Accept Improve Elapsed Remaining')  
 print('%12.2f %12.2f %s ' % \  
 (T, E, time\_string(elapsed)))  
 else:  
 remain = (self.steps - step) \* (elapsed / step)  
 print('%12.2f %12.2f %7.2f%% %7.2f%% %s %s' % \  
 (T, E, 100.0 \* acceptance, 100.0 \* improvement,  
 time\_string(elapsed), time\_string(remain)))  
  
  
def anneal(self):  
 *"""Minimizes the energy of a system by simulated annealing.  
  
 Parameters  
 state : an initial arrangement of the system  
  
 Returns  
 (state, energy): the best state and energy found.  
 """* step = 0  
 self.start = time.time()  
  
 steps = [] ### initialise a list to save the steps taken by the algorithm to find a good solution  
  
 # Precompute factor for exponential cooling from Tmax to Tmin  
 if self.Tmin <= 0.0:  
 raise Exception('Exponential cooling requires a minimum "\  
 "temperature greater than zero.')  
 Tfactor = -math.log(self.Tmax / self.Tmin)  
  
 # Note initial state  
 T = self.Tmax  
 E = self.energy()  
 prevState = self.copy\_state(self.state)  
 prevEnergy = E  
 bestState = self.copy\_state(self.state)  
 bestEnergy = E  
 trials, accepts, improves = 0, 0, 0  
 if self.updates > 0:  
 updateWavelength = self.steps / self.updates  
 self.update(step, T, E, None, None)  
  
 # Attempt moves to new states  
 while step < self.steps:  
 step += 1  
 T = self.Tmax \* math.exp(Tfactor \* step / self.steps)  
 a, b = self.move()  
 E = self.energy()  
 dE = E - prevEnergy  
 trials += 1  
 if dE > 0.0 and math.exp(-dE / T) < random.random():  
 # Restore previous state  
 self.state = self.copy\_state(prevState)  
 E = prevEnergy  
 else:  
 # Accept new state and compare to best state  
 accepts += 1  
 if dE < 0.0:  
 improves += 1  
 prevState = self.copy\_state(self.state)  
 prevEnergy = E  
  
 steps.append([a, b]) ### append the "good move" to the list of steps  
  
 if E < bestEnergy:  
 bestState = self.copy\_state(self.state)  
 bestEnergy = E  
 if self.updates > 1:  
 if step // updateWavelength > (step - 1) // updateWavelength:  
 self.update(  
 step, T, E, accepts / trials, improves / trials)  
 trials, accepts, improves = 0, 0, 0  
  
 # Return best state and energy  
 return bestState, bestEnergy, steps ### added steps to what should be returned  
  
  
def distance(a, b):  
 *"""Calculates distance between two latitude-longitude coordinates."""* R = 3963 # radius of Earth (miles)  
 lat1, lon1 = math.radians(a[0]), math.radians(a[1])  
 lat2, lon2 = math.radians(b[0]), math.radians(b[1])  
 return math.acos(math.sin(lat1) \* math.sin(lat2) +  
 math.cos(lat1) \* math.cos(lat2) \* math.cos(lon1 - lon2)) \* R  
  
  
class TravellingSalesmanProblem(Annealer):  
  
  
 *"""Test annealer with a travelling salesman problem.  
 """* # pass extra data (the distance matrix) into the constructor  
 def \_init\_(self, state, distance\_matrix):  
 self.distance\_matrix = distance\_matrix  
  
  
 super(TravellingSalesmanProblem, self).\_init\_(state) # important!  
  
  
 def move(self):  
 *"""Swaps two cities in the route."""* a = random.randint(0, len(self.state) - 1)  
 b = random.randint(0, len(self.state) - 1)  
 self.state[a], self.state[b] = self.state[b], self.state[a]  
 return a, b ### return the change made  
  
  
 def energy(self):  
 *"""Calculates the length of the route."""* e = 0  
 for i in range(len(self.state)):  
 e += self.distance\_matrix[self.state[i - 1]][self.state[i]]  
 return e  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
  
 # latitude and longitude for the six cities  
 cities = {  
 'Islamabad': (33.73, 73.08),  
 'Peshawar': (34.02, 71.56),  
 'Lahore': (31.58, 74.32),  
 'Karachi': (24.85, 67.00),  
 'Quetta': (30.18, 66.99),  
 'Faisalabad': (31.34, 73.42)  
 }  
  
 # initial state, a randomly-ordered itinerary  
 init\_state = list(cities.keys())  
 random.shuffle(init\_state)  
 reconstructed\_state = init\_state  
  
 # create a distance matrix  
 distance\_matrix = {}  
 for ka, va in cities.items():  
 distance\_matrix[ka] = {}  
 for kb, vb in cities.items():  
 if kb == ka:  
 distance\_matrix[ka][kb] = 0.0  
 else:  
 distance\_matrix[ka][kb] = distance(va, vb)  
  
 tsp = TravellingSalesmanProblem(init\_state, distance\_matrix)  
 # since our state is just a list, slice is the fastest way to copy  
 tsp.copy\_strategy = "slice"  
 state, e, steps = tsp.anneal()  
  
 while state[0] != 'Islamabad':  
 state = state[1:] + state[:1] # rotate NYC to start  
 print("Results:")  
 for city in state:  
 print("\t", city)  
  
 ### recontructed the annealing process  
 print("")  
 print("nbr. of steps:", len(steps))  
 print("Reconstructed results:")  
 for s in steps:  
 reconstructed\_state[s[0]], reconstructed\_state[s[1]] = reconstructed\_state[s[1]], reconstructed\_state[s[0]]  
 while reconstructed\_state[0] != 'Islamabad':  
 reconstructed\_state = reconstructed\_state[1:] + reconstructed\_state[:1] # rotate NYC to start  
 for city in reconstructed\_state:  
 print("\t", city)

**output:**

Text

Description automatically generated