**TASK 1:**

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

**CODE:**

*Owned*

\_\_author\_\_ *=* "Qaiser Abbas"

\_\_copyright\_\_ *=* "Copyright 2020, Artificial Intelligence lab-09"

\_\_email\_\_ *=* "qaiserabbas889@yahoo.com"

*#===============================================================*

*# {code}*

*import* math

*import* random

*import* visualize\_tsp

*import* matplotlib.pyplot *as* plt

*class* SimAnneal(object):

*def* \_\_init\_\_(self, coords, T*=-*1, alpha*=-*1, stopping\_T*=-*1, stopping\_iter*=-*1):

*self*.coords *=* coords

*self*.N *=* len(coords)

*self*.T *=* math.sqrt(*self*.N) *if* T *==* *-*1 *else* T

*self*.T\_save *=* *self*.T  *# save inital T to reset if batch annealing is used*

*self*.alpha *=* 0.995 *if* alpha *==* *-*1 *else* alpha

*self*.stopping\_temperature *=* 1e-8 *if* stopping\_T *==* *-*1 *else* stopping\_T

*self*.stopping\_iter *=* 100000 *if* stopping\_iter *==* *-*1 *else* stopping\_iter

*self*.iteration *=* 1

*self*.nodes *=* [i *for* i *in* range(*self*.N)]

*self*.best\_solution *=* None

*self*.best\_fitness *=* float("Inf")

*self*.fitness\_list *=* []

*def* *initial\_solution*(self):

        """

        Greedy algorithm to get an initial solution (closest-neighbour).

        """

        cur\_node *=* random.choice(*self*.nodes)  *# start from a random node*

        solution *=* [cur\_node]

        free\_nodes *=* set(*self*.nodes)

        free\_nodes.remove(cur\_node)

*while* free\_nodes:

            next\_node *=* min(free\_nodes, key*=lambda* x: *self*.dist(cur\_node, x))  *# nearest neighbour*

            free\_nodes.remove(next\_node)

            solution.append(next\_node)

            cur\_node *=* next\_node

        cur\_fit *=* *self*.fitness(solution)

*if* cur\_fit *<* *self*.best\_fitness:  *# If best found so far, update best fitness*

*self*.best\_fitness *=* cur\_fit

*self*.best\_solution *=* solution

*self*.fitness\_list.append(cur\_fit)

*return* solution, cur\_fit

*def* *dist*(self, node\_0, node\_1):

        """

        Euclidean distance between two nodes.

        """

        coord\_0, coord\_1 *=* *self*.coords[node\_0], *self*.coords[node\_1]

*return* math.sqrt((coord\_0[0] *-* coord\_1[0]) *\*\** 2 *+* (coord\_0[1] *-* coord\_1[1]) *\*\** 2)

*def* *fitness*(self, solution):

        """

        Total distance of the current solution path.

        """

        cur\_fit *=* 0

*for* i *in* range(*self*.N):

            cur\_fit *+=* *self*.dist(solution[i *%* *self*.N], solution[(i *+* 1) *%* *self*.N])

*return* cur\_fit

*def* *p\_accept*(self, candidate\_fitness):

        """

        Probability of accepting if the candidate is worse than current.

        Depends on the current temperature and difference between candidate and current.

        """

*return* math.exp(*-*abs(candidate\_fitness *-* *self*.cur\_fitness) */* *self*.T)

*def* *accept*(self, candidate):

        """

        Accept with probability 1 if candidate is better than current.

        Accept with probabilty p\_accept(..) if candidate is worse.

        """

        candidate\_fitness *=* *self*.fitness(candidate)

*if* candidate\_fitness *<* *self*.cur\_fitness:

*self*.cur\_fitness, *self*.cur\_solution *=* candidate\_fitness, candidate

*if* candidate\_fitness *<* *self*.best\_fitness:

*self*.best\_fitness, *self*.best\_solution *=* candidate\_fitness, candidate

*else*:

*if* random.random() *<* *self*.p\_accept(candidate\_fitness):

*self*.cur\_fitness, *self*.cur\_solution *=* candidate\_fitness, candidate

*def* *anneal*(self):

        """

        Execute simulated annealing algorithm.

        """

*# Initialize with the greedy solution.*

*self*.cur\_solution, *self*.cur\_fitness *=* *self*.initial\_solution()

        print("Starting annealing.")

*while* *self*.T *>=* *self*.stopping\_temperature *and* *self*.iteration *<* *self*.stopping\_iter:

            candidate *=* list(*self*.cur\_solution)

            l *=* random.randint(2, *self*.N *-* 1)

            i *=* random.randint(0, *self*.N *-* l)

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

*self*.accept(candidate)

*self*.T *\*=* *self*.alpha

*self*.iteration *+=* 1

*self*.fitness\_list.append(*self*.cur\_fitness)

        print("Best fitness obtained: ", *self*.best\_fitness)

        improvement *=* 100 *\** (*self*.fitness\_list[0] *-* *self*.best\_fitness) */* (*self*.fitness\_list[0])

        print(*f*"Improvement over greedy heuristic: {improvement *: .2f*}%")

*def* *batch\_anneal*(self, times*=*10):

        """

        Execute simulated annealing algorithm `times` times, with random initial solutions.

        """

*for* i *in* range(1, times *+* 1):

            print(*f*"Iteration {i}/{times} -------------------------------")

*self*.T *=* *self*.T\_save

*self*.iteration *=* 1

*self*.cur\_solution, *self*.cur\_fitness *=* *self*.initial\_solution()

*self*.anneal()

*def* *visualize\_routes*(self):

        """

        Visualize the TSP route with matplotlib.

        """

        visualize\_tsp.plotTSP([*self*.best\_solution], *self*.coords)

*def* *plot\_learning*(self):

        """

        Plot the fitness through iterations.

        """

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

        plt.ylabel("Fitness")

        plt.xlabel("Iteration")

        plt.show()

**OUTPUT:**

