Distance matrix of the problem

	City01	City02	City03	City04	City05	City06	City07	City08
City01	0	46	35	80	89	56	84	57
City02	46	0	66	64	54	87	76	85
City03	35	66	0	75	20	43	21	76
City04	80	64	75	0	33	16	68	81
City05	89	54	20	33	0	16	52	10
City06	56	87	43	16	16	0	68	22
City07	84	76	21	68	52	68	0	57
City08	57	85	76	81	10	22	57	0

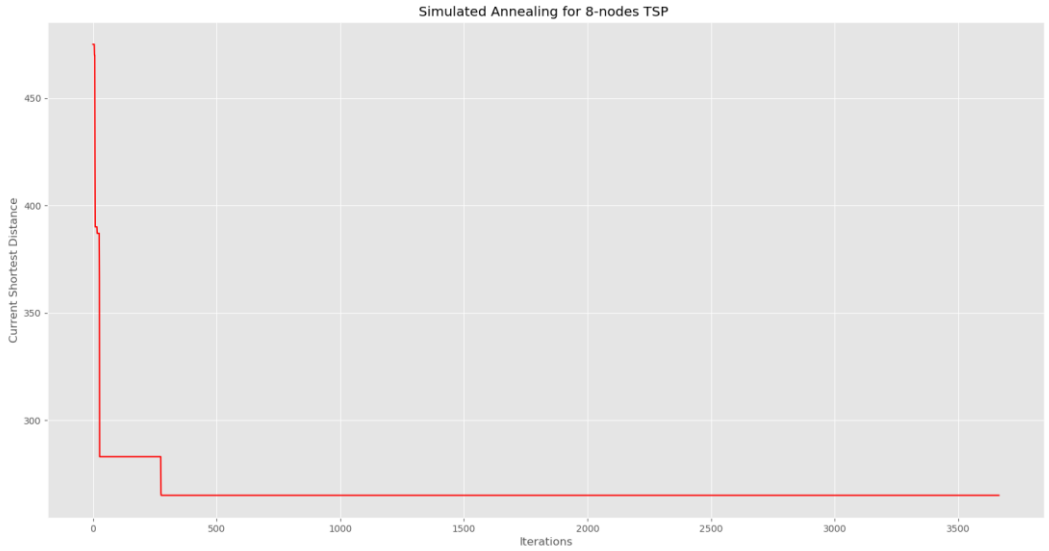
Simulated Annealing for the problem

Parameter Setting & Mechanism Selection	
Initial Temperature	500
Final Temperature	0.1
Cool Schedule	Lundy and Mees (1986) $T(k) = T(k-1) / (1+\gamma T(k-1))$ ，$\gamma = 0.001$
Number of Iterations	Linearly Increasing $n_{iter} = \alpha(t0-t)+\beta$; $\alpha = 2$, $\beta = 15$
Permutation Neighborhood	one-third opportunity to do Inversion / Transposition / Displacement respectively
Reheating	When a move accepted: $t = t / (1+\beta)$ ； $\beta = 0.2$
	When a move rejected: $t = t / (1-\gamma)$; $\gamma = 0.1$

Optimization result

Solution Under Exhaustive Enumeration	
Shortest Path:	[1, 2, 4, 6, 5, 8, 7, 3, 1]
Shortest Distance:	265
Number of Total Solutions	5040
Solution Under Simulated Annealing	
Find the shortest path: [1, 2, 4, 6, 5, 8, 7, 3, 1], with shortest distance 265 On the 276th iterations	
Number of Total Iterations:	3665

The evolution history



Program Code:

1. *Generate the distance matrix for problem*

```
"""
Simulated Annealing for TSP
By @Aaron YuKu Chen
"""
import pandas as pd
from matplotlib import pyplot as plt
import numpy as np
import random
import math

num_of_city = 8

# Generate the random cities locations
def initial_cityLocation():  # define a function to generate Distance matrix of the problem
    zero_distance = np.zeros((num_of_city, num_of_city), dtype=np.int)  # set all zero in the matrix at first
    for i in range(num_of_city):  # for half of the matrix, change the number of it
        for j in range(num_of_city):  # and make the i = j remain zero
            if i < j:  # compute one by one
                temp_dis = random.randint(10, 90)  # generate the distance between 10 and 90
                zero_distance[i][j], zero_distance[j][i] = temp_dis, temp_dis  # copy it to the symmetry one
    return zero_distance  # return the distance to this function

cities_Location = initial_cityLocation()  # generate the initial cities locations
print(cities_Location)  # print out the distance matrix
```

2. *Determine a distance matrix for my problem.*

```
# using the GIVEN DISTANCE
cities_Location = [[0, 46, 35, 80, 89, 56, 84, 57],  # I selected this distance matrix as my problem
                   [46, 0, 66, 64, 54, 87, 76, 85],
                   [35, 66, 0, 75, 20, 43, 21, 76],
                   [80, 64, 75, 0, 33, 16, 68, 81],
                   [89, 54, 20, 33, 0, 16, 52, 10],
                   [56, 87, 43, 16, 16, 0, 68, 22],
                   [84, 76, 21, 68, 52, 68, 0, 57],
                   [57, 85, 76, 81, 10, 22, 57, 0]]
cities_Location = np.array(cities_Location)  # generate the initial cities locations
print(cities_Location)  # print out the distance matrix
```

3. *Write the exhaustive enumeration*

To find the optimal shortest path and their distance solution for my problem.

```
# the exhaustive enumeration
sum_temp = 1  # this part wants to compute the number of total solutions for n city under exhaustive enumeration
for i in range(1, num_of_city):
    sum_temp *= i  # compute the n factorial
print("Number of EXPECT Solutions: ", sum_temp)  # print the number of total solutions for exhaustive enumeration

the_shortest_distance = 1000  # set a temporary shortest distance for compute later
the_count_time = 0  # check the number of iterations for exhaustive enumeration
the_shortest_path = []  # set a temporary path for use it later
for leg_01 in range(1, num_of_city):  # for the first leg, compute each potential  city
    the_path_leg01 = [0]  # record the leg01 visited
    distance_part01 = cities_Location[0][leg_01]  # compute the distance leg01 visited
    the_path_leg01.append(leg_01)  # record the leg01 visited
    for leg_02 in range(1, num_of_city):
        the_path_leg02 = [0, leg_01]
        if leg_02 < leg_01:  # avoid selected the same city
            distance_part02 = distance_part01 + cities_Location[leg_01][leg_02]
        elif leg_02 > leg_01:
            distance_part02 = distance_part01 + cities_Location[leg_01][leg_02]
        else:
            continue
        the_path_leg02.append(leg_02)  # record the leg02 visited
        for leg_03 in range(1, num_of_city):
            the_path_leg03 = [0, leg_01, leg_02]  # record city already visited so far
            if leg_03 < leg_02 and leg_03 != leg_01:  # avoid selected same city and the city already visited
```

```
    for leg_03 in range(1, num_of_city):
        the_path_leg03 = [0, leg_01, leg_02]  # record city already visited so far
        if leg_03 < leg_02 and leg_03 != leg_01:  # avoid selected same city and the city already visited
            distance_part03 = distance_part02 + cities_Location[leg_02][leg_03]
        elif leg_03 > leg_02 and leg_03 != leg_01:
            distance_part03 = distance_part02 + cities_Location[leg_02][leg_03]
        else:
            continue
        the_path_leg03.append(leg_03)  # record the city already visited
        for leg_04 in range(1, num_of_city):  # compute the leg04
            the_path_leg04 = [0, leg_01, leg_02, leg_03]
            if leg_04 < leg_03 and leg_04 != leg_01 and leg_04 != leg_02:
                distance_part04 = distance_part03 + cities_Location[leg_03][leg_04]
            elif leg_04 > leg_03 and leg_04 != leg_01 and leg_04 != leg_02:
                distance_part04 = distance_part03 + cities_Location[leg_03][leg_04]
            else:
                continue
            the_path_leg04.append(leg_04)
            for leg_05 in range(1, num_of_city):  # compute the leg05
                the_path_leg05 = [0, leg_01, leg_02, leg_03, leg_04]
                if leg_05 < leg_04 and leg_05 != leg_01 and leg_05 != leg_02 and leg_05 != leg_03:
                    distance_part05 = distance_part04 + cities_Location[leg_04][leg_05]
                elif leg_05 > leg_04 and leg_05 != leg_01 and leg_05 != leg_02 and leg_05 != leg_03:
                    distance_part05 = distance_part04 + cities_Location[leg_04][leg_05]
```

```
else:
    continue
the_path_leg05.append(leg_05)  # record the city visited so far
for leg_06 in range(1, num_of_city):
    the_path_leg06 = [0, leg_01, leg_02, leg_03, leg_04, leg_05]
    if leg_06 < leg_05 and leg_06 != leg_01 and leg_06 != leg_02 and leg_06 != leg_03 and leg_06 != leg_04:
        distance_part06 = distance_part05 + cities_Location[leg_05][leg_06]
    elif leg_06 > leg_05 and leg_06 != leg_01 and leg_06 != leg_02 and leg_06 != leg_03 and leg_06 != leg_04:
        distance_part06 = distance_part05 + cities_Location[leg_05][leg_06]
    else:
        continue
    the_path_leg06.append(leg_06)
    for leg_07 in range(1, num_of_city):  # compute the leg07
        the_path_leg07 = [0, leg_01, leg_02, leg_03, leg_04, leg_05, leg_06]
        if leg_07 < leg_06 and leg_07 != leg_01 and leg_07 != leg_02 and leg_07 != leg_03 and leg_07 != leg_04 and leg_07 != leg_05:
            distance_part07 = distance_part06 + cities_Location[leg_06][leg_07]
        elif leg_07 > leg_06 and leg_07 != leg_01 and leg_07 != leg_02 and leg_07 != leg_03 and leg_07 != leg_04 and leg_07 != leg_05:
            distance_part07 = distance_part06 + cities_Location[leg_06][leg_07]
        else:
            continue
        the_path_leg07.append(leg_07)  # record the city already visited
        for leg_08 in range(num_of_city):
            the_path_leg08 = [0, leg_01, leg_02, leg_03, leg_04, leg_05, leg_06, leg_07]
            if leg_08 != leg_07 and leg_08 != leg_06 and leg_08 != leg_05 and leg_08 != leg_04 and leg_08 != leg_03 and leg_08 != leg_02 and leg_08 != leg_01:
                distance_part08 = distance_part07 + cities_Location[leg_07][0]
            else:
```

```
                else:
                    continue
                the_path_leg08.append(leg_08)
                # all add one make it into the real city
                the_total_path = [x + 1 for x in the_path_leg08]
                the_count_time += 1
                if distance_part08 < the_shortest_distance:  # compare which one is shortest
                    the_shortest_distance = distance_part08
                    the_shortest_path = the_total_path

print('Shortest Path: ', the_shortest_path, 'and their distance: ', the_shortest_distance)
print('-------------------------------------------------')
```

4. *Start the Simulated Annealing Algorithm*

First, establish a function to generate an initial path randomly.

```
# For the Simulated Annealing part

def initial_sol():  # generate a random initial solution
    movement_candi = [2, 3, 4, 5, 6, 7, 8]  # made a list prepare to random choice
    random_list = random.sample(movement_candi, 7)  # made these city sequence random
    random_list.insert(0, 1)  # add the first city
    random_list.insert(num_of_city, 1)  # add the final trip to go back to city 01
    return random_list
```

5. *Permutation Neighborhood by Inversion / Transposition/ Displacement.*

```
# Mechanism Permutation Neighborhood by Inversion / Transposition / Displacement
def select_neighborhood(the_path):
    k = random.random()  # generate a number to choice what mechanism of select neighborhood this time
    if k > 0.66:  # one-third opportunity to do Inversion
        rand_Loc = random.randint(2, 7)  # choice a location
        temp_num = the_path[rand_Loc]  # chosen location's value
        the_path[rand_Loc] = the_path[rand_Loc - 1]  # transfer the value to previous one
        the_path[rand_Loc - 1] = temp_num  # also give the value to exchange one
    elif 0.66 > k > 0.33:  # one-third opportunity to do Transposition
        rand_Loc01, rand_Loc02 = random.randint(1, 7), random.randint(1, 7)  # choice 2 locations
        if rand_Loc01 != rand_Loc02:  # if that 2 location different
            temp_num = the_path[rand_Loc01]
            the_path[rand_Loc01] = the_path[rand_Loc02]  # make them change each other
            the_path[rand_Loc02] = temp_num
        else:  # if the first outcome are the same
            temp_list = [2, 3, 4, 5, 6, 7]  # once they are the same number
            temp_list.pop(rand_Loc02 - 2)  # re-choice a location again
            rand_Loc01 = random.choice(temp_list)
            temp_num = the_path[rand_Loc01]
            the_path[rand_Loc01] = the_path[rand_Loc02]  # make them change each other
            the_path[rand_Loc02] = temp_num
    else:  # one-third opportunity to do Displacement
        rand_Loc01, rand_Loc02 = random.randint(1, 7), random.randint(1, 7)  # chose two location
        temp_num = the_path[rand_Loc01]
        the_path.pop(rand_Loc01)  # make the chosen one insert into another location
        the path.insert(rand Loc02, temp num)

    else:  # one-third opportunity to do Displacement
        rand_Loc01, rand_Loc02 = random.randint(1, 7), random.randint(1, 7)  # chose two location
        temp_num = the_path[rand_Loc01]
        the_path.pop(rand_Loc01)  # make the chosen one insert into another location
        the_path.insert(rand_Loc02, temp_num)
    return the_path  # return a new path
```

6. *Established three functions.*

find distance: make a path could show their distance in my problem.

delta: compute the distance current one and previous one.

accept worse moves: once occurred a worse move, compute the move probability and generate a random number.

```
def find_distance(someone_path):  # this function make the path could got its distance
    distance = 0
    for city in range(0, num_of_city):  # city by city to find out its distance
        distance += cities_Location[someone_path[city] - 1][someone_path[city + 1] - 1]
    return distance


def delta(compute_distance_maybe_update, compute_distance_current):  # compute the delta value for two distance
    the_delta = compute_distance_maybe_update - compute_distance_current
    return the_delta


def accept_worse_moves(compute_delta_value, compute_temperature_current, compute_path_maybe_update,
                       compute_distance_maybe_update, compute_curr_path, compute_curr_dis):
    r = random.random()  # if it is a worse move, also make a chance let it accepted
    move_prob = math.exp(- compute_delta_value / compute_temperature_current)  # move prob formula
    if r > move_prob:  # accepted this worse move
        compute_path_current = compute_path_maybe_update
        compute_distance_current = compute_distance_maybe_update
    else:  # do not accepted this worse
        compute_path_current = compute_curr_path
        compute_distance_current = compute_curr_dis
    return compute_path_current, compute_distance_current
```

7. *Parameter Setting, Generate initial path and its distance.*

```
# Parameter Setting
initTemp = 500  # set the starting temperature as 500
finalTemp = 1  # set the final temperature as 1
rate_of_cooling = 0.001  # the rate of cooling
a_of_LinearlyIncreasing = 2
b_of_LinearlyIncreasing = 15
b_of_Reheating = 0.2
r_of_Reheating = 0.1

initPath = initial_sol()  # realize the initial Path
initDis = find_distance(initPath)
# print('Initial Path is: ', path_current, " , and their distance is: ", distance_current)
total_Iteration = 0  # set the initial iteration
shortestDis = 500  # set the initial shortest distance
shortestPath = [1, 2, 3, 4, 5, 6, 7, 8, 1]  # set the initial shortest Path
currPath = initPath  # this make initial path equal to current path
currDis = initDis  # set the initial distance equal to current distance
currTemp, candidateTemp = initTemp, initTemp
shortest_DisList = []
```

8. *Start the structure of Simulated Annealing*

```
# structure of simulated annealing algorithms
while currTemp > finalTemp:  # once currently temperature smaller than final temperature , stop this algorithm
    # Number of Iterations using the Linearly increasing
    limitIterations_inTemp = a_of_LinearlyIncreasing * (currTemp - candidateTemp) + b_of_LinearlyIncreasing
    limitIterations_inTemp = int(limitIterations_inTemp)
    iterations = 0
    currTemp = candidateTemp
    while iterations < limitIterations_inTemp:  # set the number of iterations in each temperature
        candidatePath = select_neighborhood(currPath)  # select the neighborhood, find the candidate Path
        candidateDis = find_distance(candidatePath)  # find the candidate distance
        delta_value = delta(candidateDis, currDis)
        if delta_value < 0:  # if the distance become smaller
            if currDis < shortestDis:  # if shortest distance could update
                shortestDis = currDis
                shortestPath = currPath
                print('the current shortest path: ', shortestPath, 'the current shortest distance: ',
                      shortestDis, ' and this is ', total_Iteration, ' iterations')
                # to record for the  convergence history
            else:
                currDis, currPath = currDis, currPath
            currPath = candidatePath
            currDis = candidateDis
            # cooling schedule using the model by Lundy and Mees(1986)
            candidateTemp = currTemp / (1 + (rate_of_cooling * currTemp))
            # Reheating when move is accepted
            candidateTemp = candidateTemp / (1 + b of Reheating)

        else:  # this part write the condition of  "may accept worse moves"
            if currDis < shortestDis:
                shortestDis = currDis
                shortestPath = currPath
                print('the current shortest path: ', shortestPath, 'the current shortest distance: ',
                      shortestDis, ' and this is ', total_Iteration, ' iterations')
                # to record for the convergence history

            else:
                currDis, currPath = currDis, currPath
            # accepted the worse moves or not
            currPath, currDis = accept_worse_moves(delta_value, currTemp, candidatePath,
                                                   candidateDis, currPath, currDis)
            # cooling schedule using the model by Lundy and Mees(1986)
            candidateTemp = currTemp / (1 + (rate_of_cooling * currTemp))
            # Reheating when move is rejected
            candidateTemp = candidateTemp / (1 - r_of_Reheating)

        iterations += 1  # finish this iteration
        total_Iteration += 1  # record the total iterations so far
        shortest_DisList.append(shortestDis)   # record the shortest distance
    # print('current path: ', currPath, ', and their distance: ', currDis)

print('Currently total iterations: ', total_Iteration)   # in the end , print the total iteration of this algorithm
print('------------------------------------------------------------')
```

9. *Visualization*

```
# visualization the performance for this algorithm
print("Shortest distance(y): ", shortestDis)  # print the final shortest path
# print("Optima PATH:", shortestPath)
shortest_distance = pd.DataFrame(shortest_DisList)
plt.style.use('ggplot')
plt.plot(range(total_Iteration), shortest_distance, 'r')
plt.xlabel("Iterations")
plt.ylabel("Current Shortest Distance")
plt.title("Simulated Annealing for 8-nodes TSP ")
# plt.figure()
plt.show()
```