Bahria University

Software Engineering Department

Course: SEL-221 Artificial Intelligence Lab
Term: Fall 2020, Class: BSE 5(B)

Assignment No:

0	2

Submitted By:

(Name) Qaiser Abbas (Reg. No.) 57245

Submission Date							
2	4	/	1	2	/	2	0

(Date: DD/MM/YY)

Submitted To:

Engr. M. Rehan Baig

(Subject Teacher)

Signature: ________________ Max Marks: ___________ Marks Obtained: _____________

Contents

Question: 3
Breadth First Search and Depth First Search: 4
Uniform Cost Search: 8
A* Search: 9
Simulated Annealing on Tkinter: 12
MIN MAX Algorithm: 15

Bahria University
Discovering Knowledge

BAHRIA UNIVERSITY (KARACHI CAMPUS)

ASSGINMENT # 2 - FALL 2020

Artificial Intelligence Lab

Class: **BSE 5(B)**

Lab Instructor: **Engr. Muhammad Rehan Baig** **Submission Deadline: 31th December, 2020**

Max Marks: **5**

Question: Apply ***Breadth First Search***, ***Depth First Search***, ***Uniform Cost Search***, ***A* Search***, ***Simulated Annealing*** and ***MIN MAX*** **Algorithms** to Solve Sales Man Traveling problem. Identify **Time Complexity** for each algorithm and find best suited algorithm for solving this problem with stated algorithms along with complete code.

Note:

- Please Provide Proper documentation of your Assignment.
- If you submit your assignment after the given deadline then **2 Marks** will be deducted for the late submissions.
- Copied assignment will be marked **zero**.
- ***Please don't share your assignment*** with any of your colleagues Because **Same Assignments** will be marked as copied and **ZERO** will be assigned to both. Author and Copier

Breadth First Search and Depth First Search:

```
__author__ = "Qaiser Abbas"
__copyright__ = "Copyright 2020, Artificial Intelligence Assignment-02"
__email__ = "qaiserabbas889@yahoo.com"

edges = []
print("Enter the distance of All cities from City A:")
edges_cityA = []
edges_cityB = []
edges_cityC = []
edges_cityD = []
edges_cityE = []
cost = int(input())
edges_cityA.append(0)
edges_cityA.append(cost)
cost = int(input())
edges_cityA.append(cost)
cost = int(input())
edges_cityA.append(cost)
cost = int(input())
edges_cityA.append(cost)
print("Enter the distances of All cities from city B")
cost = int(input())
edges_cityB.append(cost)
edges_cityB.append(0)
cost = int(input())
edges_cityB.append(cost)
cost = int(input())
edges_cityB.append(cost)
cost = int(input())
edges_cityB.append(cost)
print("Enter the distance of All cities from city C")
cost = int(input())
edges_cityC.append(cost)
cost = int(input())
edges_cityC.append(cost)
edges_cityC.append(0)
cost = int(input())
edges_cityC.append(cost)
cost = int(input())
edges_cityC.append(cost)
print("Enter the distance of All cities from city D")
cost = int(input())
edges_cityD.append(cost)
cost = int(input())
edges_cityD.append(cost)
cost = int(input())
edges_cityD.append(cost)
edges_cityD.append(0)
cost = int(input())
edges_cityD.append(cost)

print("Enter the distance of All cities from city E")
cost = int(input())
edges_cityE.append(cost)
cost = int(input())
```

```
edges_cityE.append(cost)
cost = int(input())
edges_cityE.append(cost)
cost = int(input())
edges_cityE.append(cost)
edges_cityE.append(0)

edges.append(edges_cityA)
edges.append(edges_cityB)
edges.append(edges_cityC)
edges.append(edges_cityD)
edges.append(edges_cityE)


def TSP_bfs(edges):
    q = []
    path = []
    visited = [False] * 6
    p = [0]
    q.append((0, 0, visited, p))
    while len(q) != 0:
        cnt = 0
        curr = q.pop(0)
        curr[2][curr[0]] = True
        for i in range(5):
            if curr[2][i] == False:
                cnt += 1
        if cnt == 0:
            P = curr[3]
            P.append(0)
            path.append((curr[1] + edges[curr[0]][0], P))
        for i in range(5):
            if curr[2][i] == False:
                tmp = [False] * 6
                for j in range(5):
                    tmp[j] = curr[2][j]
                P = []
                for j in range(len(curr[3])):
                    P.append(curr[3][j])
                P.append(i)
                q.append((i, curr[1] + edges[curr[0]][i], tmp, P))

    mini = 1000
    P = []
    for i in range(len(path)):
        if mini > path[i][0]:
            mini = path[i][0]
            P = path[i][1]
        return mini, P
print('******BFS Solution******')
print(TSP_bfs(edges))
```

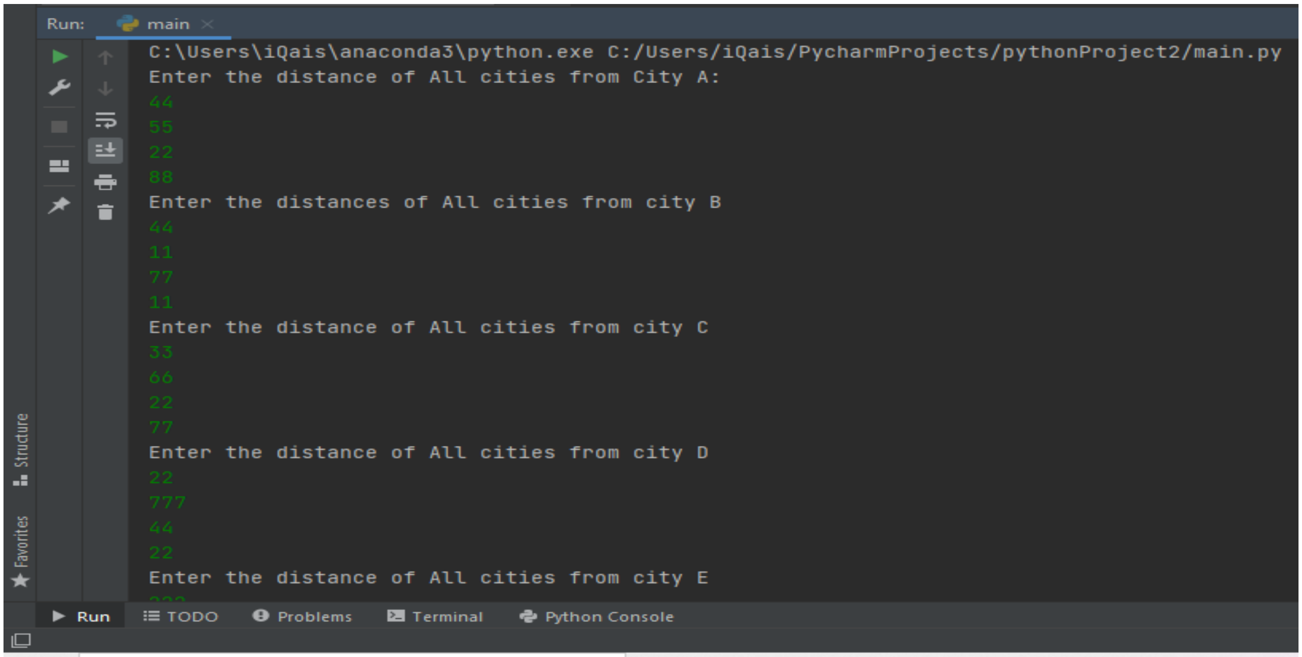
```python
def TSP_dfs(node, edges, visited, cost, path):
    cnt = 0
    path.append(node)
    visited[node] = True
    for i in range(5):
        if visited[i] == False:
            cnt += 1
    if cnt == 0:
        path.append(0)
        return (cost + edges[node][0]), path
    mini = 10000
    A = []
    for i in range(5):
        if visited[i] == False:
            tmp = [False]*6
            for j in range(5):
                tmp[j] = visited[j]
            P = []
            for j in range(len(path)):
                P.append(path[j])
            t, l = TSP_dfs(i, edges, tmp, cost + edges[node][i], P)
            if mini > t:
                mini = t
                A = l
    return mini, A

visited = [False]*6
path = []
print('******DFS Solution******')
print(TSP_dfs(0, edges, visited, 0, path))
```

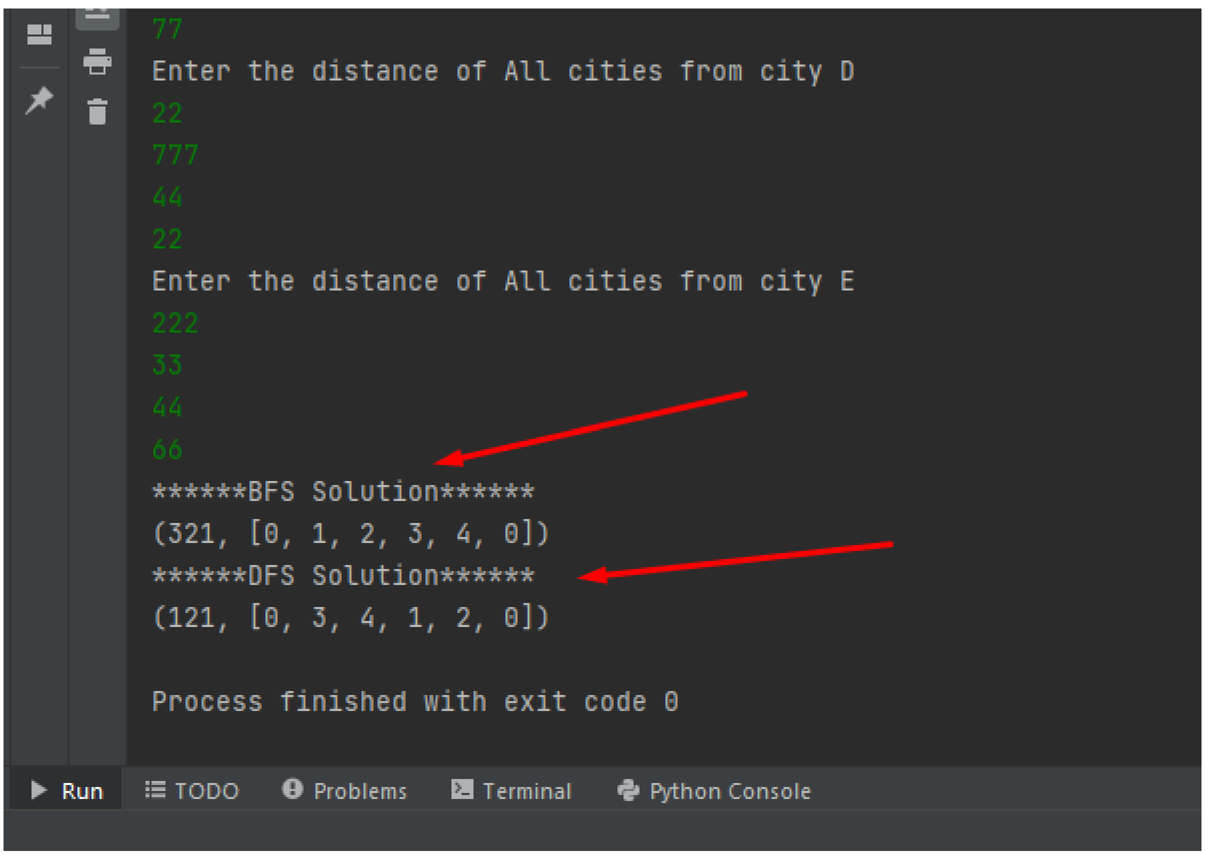
Time complexity of BFS: O(b^ d)

Time complexity of DFS: O(b^ m)

OUTPUT:



```
C:\Users\iQais\anaconda3\python.exe C:/Users/iQais/PycharmProjects/pythonProject2/main.py
Enter the distance of All cities from City A:
44
55
22
88
Enter the distances of All cities from city B
44
11
77
11
Enter the distance of All cities from city C
33
66
22
77
Enter the distance of All cities from city D
22
777
44
22
Enter the distance of All cities from city E
```

```
77
Enter the distance of All cities from city D
22
777
44
22
Enter the distance of All cities from city E
222
33
44
66
******BFS Solution******
(321, [0, 1, 2, 3, 4, 0])
******DFS Solution******
(121, [0, 3, 4, 1, 2, 0])

Process finished with exit code 0
```

Uniform Cost Search:

CODE:

```
# Owned
__author__ = "Qaiser Abbas"
__copyright__ = "Copyright 2020, Artificial Intelligence lab-06"
__email__ = "qaiserabbas889@yahoo.com"
#===============================================================
# {code}
import queue as Q
def search(graph, start, end):
    whileiterations = 0
    foriteration = 0
    if start not in graph:
        raise TypeError(str(start) + ' not found in graph !')
    if end not in graph:
        raise TypeError(str(end) + ' not found in graph !')
    queue = Q.PriorityQueue()
    queue.put((0, [start]))
    while not queue.empty():
        whileiterations = whileiterations+1
        node = queue.get()
        current = node[1][len(node[1]) - 1]


        cost = node[0]
        for neighbor in graph[current]:
            foriteration = foriteration+1
            temp = node[1][:]
            temp.append(neighbor)
            queue.put((cost + graph[current][neighbor], temp))
def main():
    graph = {
    'A': {'B': 75, 'C': 118, 'D': 140, 'E': 131},
    'B': {'C': 120, 'D': 175, 'E': 110, 'A': 39},
    'C': {'D': 110, 'E': 241, 'A': 29, 'B': 180},
    'D': {'E': 130, 'A': 111, 'B': 99, 'C': 60},
    'E': {'A': 190, 'B': 151, 'C': 199, 'D': 180},
    }

main()
```

Time complexity of UCS: Let C^* is Cost of the optimal solution, and **ε** is each step to get closer to the goal node. Then the number of steps is = $C^*/\varepsilon+1$. Here we have taken +1, as we start from state 0 and end to C^*/ε. = $\mathbf{O(b^{1 + [C^*/\varepsilon]})}$

OUTPUT:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore

PS C:\Users\iQais> & C:/Users/iQais/AppData/Local/Programs/
ling_salesman_problem.py"
Path Found: A C B E D A  Total Cost 263
PS C:\Users\iQais>
```

A* Search:

```python
# Owned
__author__ = "Qaiser Abbas"
__copyright__ = "Copyright 2020, Artificial Intelligence Assignment-02"
__email__ = "qaiserabbas889@yahoo.com"
#===============================================================

import random

class TSP(object):

    def getDistance(self, P1, P2):
        """
            Generates distance between 2 points
        """
        self.P1 = P1
        self.P2 = P2
        distance = ((self.P1[0] - self.P2[0]) ** 2 + (self.P1[1] -
self.P2[1]) ** 2) ** (1 / 2)
        return distance

    def generateCoordinates(self):
        """
            This function generates random coordinates for cities
```

```
        Returns
        -------
        list
            Returns x and y coordinates in the list.

        """
        x = random.randint(0, 101)
        y = random.randint(0, 101)
        return [x, y]

    def calculateDist(self, s, n):
        """
            Calculates the total distance in a state, eg [0,1,2,3]
        """
        self.calD = s
        self.nu = n
        dist = 0
        total = 0
        for i in range(self.nu):
            xi = self.calD[i]
            yj = self.calD[i + 1]
            dist = self.getDistance(self.coord[xi], self.coord[yj])
            total += dist
        return total

    def hue(self, chosenList, cityList):
        """
            Generates the total heuristic plus path cost and sends it out
        """
        toCheck = cityList[:]
        SPL = chosenList[:]
        dl = 999999999
        fCost = []
        l = []

        for i in chosenList:
            toCheck.remove(i)

        for e in toCheck:
            l.clear()
            l.append(e)
            rest = cityList[:]
            totalDist = 0

            while len(rest) > 0:
                dl = 99999
                for i in l:
                    if i in rest:
                        rest.remove(i)

                for n in l:
                    for m in rest:
                        d = self.getDistance(self.coord[n], self.coord[m])
                        if d < dl:
                            dl = d
                            c = m
```

```
                if c not in l:
                    l.append(c)

                totalDist += d

            g = self.getDistance(self.coord[e], self.coord[SPL[-1]])
            k = self.getDistance(self.coord['0'], self.coord[e])
            f = g + totalDist + k

            fCost.append((f, e))

        fCost.sort()

        return (fCost[0][1])

    def solver(self):
        """
            Main function that solves the TSP and output

        Returns
        -------
        None.

        """

        cities = int(input("How many cities do you want to generate: "))

        self.coord = {}
        cityList = []
        chosenList = ['0']
        self.cities = cities

        for i in range(self.cities):
            a = str(i)
            l = self.generateCoordinates()
            self.coord[a] = l
            cityList.append(a)

        # currentState = cityList[:]

        for i in range(len(cityList) - 1):
            x = self.hue(chosenList, cityList)
            chosenList.append(x)

        final = chosenList + ['0']
        il = cityList + ['0']

        fd = self.calculateDist(final, len(final) - 1)
        id = self.calculateDist(il, len(il) - 1)

        print("\nCoordinates:")
        for each in self.coord.items():
            print('City {} has coordinate {}'.format(each[0], each[1]))

        print("\nInitial state to travel: ", il, "\n")
        print("Initial distance was %.2f km. \n" % id)
```

```
        print("Final state (optimized) to travel: ", final, "\n")
        print("Optimized distance is %.2f km. \n" % fd)


def main():
    tsp = TSP()
    tsp.solver()


if __name__ == "__main__":
    main()
```

Time complexity of A*: A* is cost-optimal, the worse case time complexity is O(E), where E is the number of edges in the graph

OUTPUT:

```
Run:  main
C:\Users\iQais\anaconda3\python.exe C:/Users/iQais/AppData/Local/Temp/tsp.py/main.py
How many cities do you want to generate: 4

Coordinates:
City 0 has coordinate [82, 79]
City 1 has coordinate [36, 88]
City 2 has coordinate [68, 54]
City 3 has coordinate [59, 63]

Initial state to travel:  ['0', '1', '2', '3', '0']

Initial distance was 134.31 km.

Final state (optimized) to travel:  ['0', '2', '3', '1', '0']

Optimized distance is 122.22 km.


Process finished with exit code 0
```

Simulated Annealing on Tkinter:

```
# Owned
__author__ = "Qaiser Abbas"
__copyright__ = "Copyright 2020, Artificial Intelligence Assignment-02"
__email__ = "qaiserabbas889@yahoo.com"
#===============================================================

import math
import random
import matplotlib.pyplot as plt
```

```
from util import City, read_cities, write_cities_and_return_them,
generate_cities, visualize_tsp, path_cost


class SimAnneal(object):
    def __init__(self, cities, temperature=-1, alpha=-1,
stopping_temperature=-1, stopping_iter=-1):
        self.cities = cities
        self.num_cities = len(cities)
        self.temperature = math.sqrt(self.num_cities) if temperature == -1
else temperature
        self.T_save = self.temperature
        self.alpha = 0.999 if alpha == -1 else alpha
        self.stopping_temperature = 1e-8 if stopping_temperature == -1 else
stopping_temperature
        self.stopping_iter = 100000 if stopping_iter == -1 else stopping_iter
        self.iteration = 1
        self.route = None
        self.best_fitness = float("Inf")
        self.progress = []
        self.cur_cost = None

    def greedy_solution(self):
        start_node = random.randint(0, self.num_cities)  # start from a
random node
        unvisited = self.cities[:]
        del unvisited[start_node]
        route = [cities[start_node]]
        while len(unvisited):
            index, nearest_city = min(enumerate(unvisited), key=lambda item:
item[1].distance(route[-1]))
            route.append(nearest_city)
            del unvisited[index]
        current_cost = path_cost(route)
        self.progress.append(current_cost)
        return route, current_cost

    def accept_probability(self, candidate_fitness):
        return math.exp(-abs(candidate_fitness - self.cur_cost) /
self.temperature)

    def accept(self, guess):
        guess_cost = path_cost(guess)
        if guess_cost < self.cur_cost:
            self.cur_cost, self.route = guess_cost, guess
            if guess_cost < self.best_fitness:
                self.best_fitness, self.route = guess_cost, guess
        else:
            if random.random() < self.accept_probability(guess_cost):
                self.cur_cost, self.route = guess_cost, guess

    def run(self):
        self.route, self.cur_cost = self.greedy_solution()
        while self.temperature >= self.stopping_temperature and
self.iteration < self.stopping_iter:
            guess = list(self.route)
            left_index = random.randint(2, self.num_cities - 1)
```

```
            right_index = random.randint(0, self.num_cities - left_index)
            guess[right_index: (right_index + left_index)] =
reversed(guess[right_index: (right_index + left_index)])
            self.accept(guess)
            self.temperature *= self.alpha
            self.iteration += 1
            self.progress.append(self.cur_cost)

        print("Best fitness obtained: ", self.best_fitness)

    def visualize_routes(self):
        visualize_tsp('simulated annealing TSP', self.route)

    def plot_learning(self):
        fig = plt.figure(1)
        plt.plot([i for i in range(len(self.progress))], self.progress)
        plt.ylabel("Distance")
        plt.xlabel("Iterations")
        plt.show(block=False)


if __name__ == "__main__":
    cities = read_cities(64)
    sa = SimAnneal(cities, stopping_iter=15000)
    sa.run()
    sa.plot_learning()
    sa.visualize_routes()
```

Data file: with 8 cities

591 917

315 81

895 990

508 595

367 539

73 728

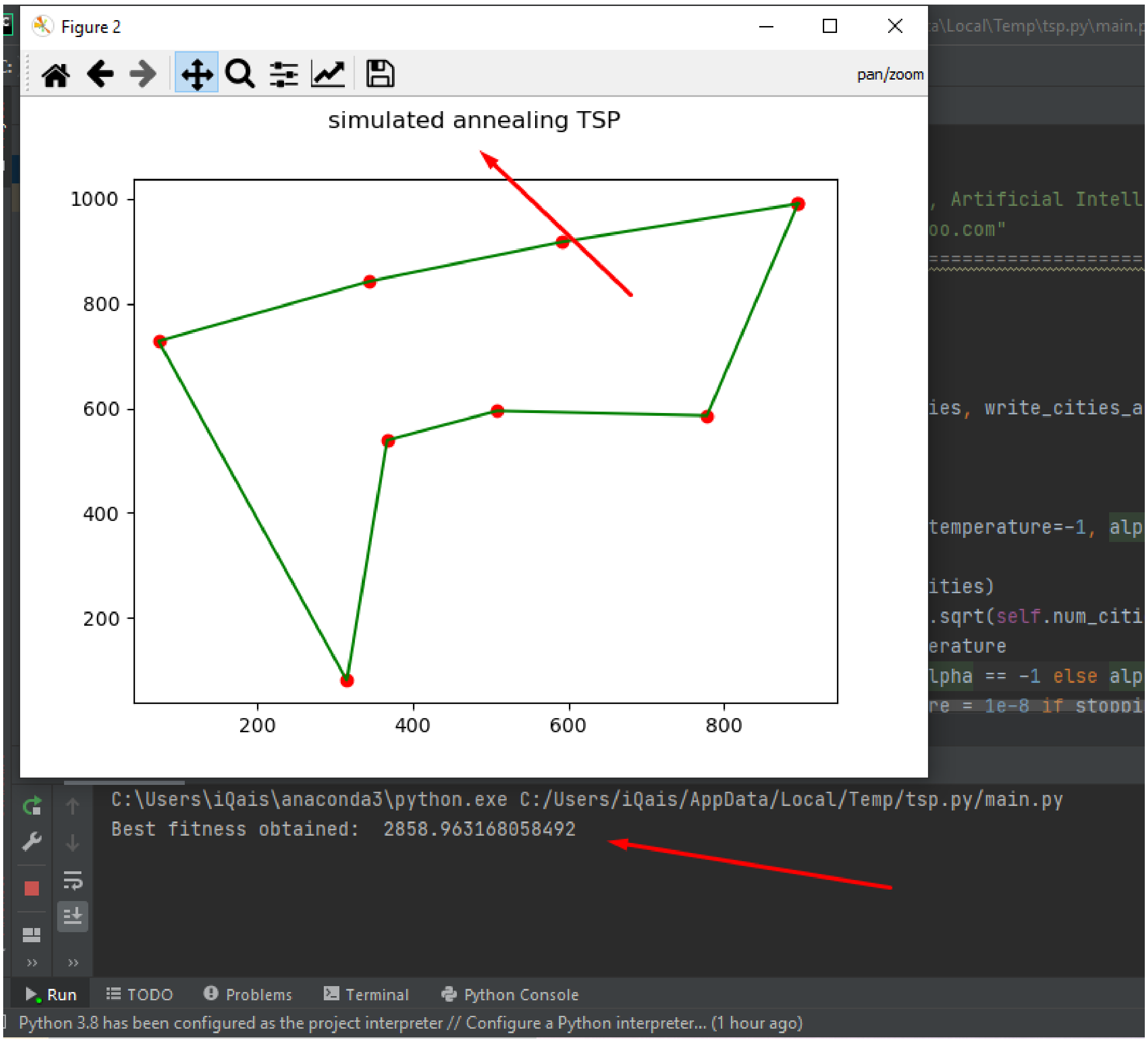
344 842

778 586

Time complexity of Simulated Annealing: if the maximum degree is bounded, the upper bound is O(v 5), where v is the number of vertices.

The best thing about simulated annealing is that it requires very less memory. It will not give optimal solution, but the solution provided by this will be good in reasonable time.

OUTPUT:



MIN MAX Algorithm:

```
__author__ = "Qaiser Abbas"
```

```
__copyright__ = "Copyright 2020, Artificial Intelligence Assignment-02"
__email__ = "qaiserabbas889@yahoo.com"
from gurobipy import *
import itertools
from math import sqrt


    def min_max_length_under_complete_graph(city_num, deliver_num, weight_metrix,
 TL):
        model = Model("TSP")
        model.setParam(GRB.Param.TimeLimit, TL)
        # Create variables
        x = {}
        for i in range(city_num):
            for j in range(city_num):
                for k in range(deliver_num):
                    x[i, j, k] = model.addVar(vtype=GRB.BINARY, name='e_' + str(i
) + '_' + str(j) + '_' + str(k))
        Q = model.addVar(name='Q')

        model.setObjective(Q, GRB.MINIMIZE)

        for k in range(deliver_num):
            model.addConstr(quicksum(x[0, j, k] for j in range(1, city_num)) == 1
)
            model.addConstr(quicksum(x[i, 0, k] for i in range(1, city_num)) == 1
)
        for i in range(1, city_num):
            model.addConstr(quicksum(x[i, j, k]
                                     for j in range(city_num)
                                     for k in range(deliver_num)) == 1
                            )
        for j in range(1, city_num):
            model.addConstr(quicksum(x[i, j, k]
                                     for i in range(city_num)
                                     for k in range(deliver_num)) == 1
                            )
        for r in range(1, city_num):
            for k in range(deliver_num):
                model.addConstr((quicksum(x[i, r, k] for i in range(city_num))
                                 - quicksum(x[r, j, k] for j in range(city_num)))
 == 0
                                )
        model.addConstrs((x[i, i, k] == 0
                          for k in range(deliver_num)
```

```
                        for i in range(1, city_num)), name='C'
                       )
        for k in range(deliver_num):
            model.addConstr(quicksum(weight_metrix[i][j] * x[i, j, k]
                                     for i in range(city_num)
                                     for j in range(city_num)) <= Q)


        # Callback - use lazy constraints to eliminate sub-tours
        def subtourelim(model, where):
            if where == GRB.callback.MIPSOL:
                # make a list of edges selected in the solution
                for k in range(deliver_num):
                    selected = []
                    visited = set()
                    for i in range(city_num):
                        sol = model.cbGetSolution([x[i, j, k] for j in range(city
_num)])
                        new_selected = [(i, j) for j in range(city_num) if sol[j]
 > 0.5]
                        selected += new_selected


                        if new_selected:
                            visited.add(i)


                    tour = subtour(selected, visited)


                    if len(tour) < len(visited):
                        # add a subtour elimination constraint
                        expr = quicksum(x[i, j, k] for i, j in itertools.permutat
ions(tour, 2))
                        model.cbLazy(expr <= len(tour) - 1)


        # Optimize model
        model.update()
        model.params.LazyConstraints = 1
        # model.optimize()
        model.optimize(subtourelim)
```

```
        node_mat = [[[0 for i in range(city_num)] for i in range(city_num)] for i
in range(deliver_num)]
        for k in range(deliver_num):
            for i in range(city_num):
                for j in range(city_num):
                    node_mat[k][i][j] = x[i, j, k].x


        allpath = []
        for k in range(deliver_num):
            path = []
            cnt = 0
            while True:
                path.append(cnt)
                for j in range(city_num):
                    if node_mat[k][cnt][j] > 0.5:
                        cnt = j
                        break
                if cnt in path:
                    path.append(cnt)
                    break
            allpath.append(path)
        return allpath


    def subtour(edges, visited):
        unvisited = list(visited)
        cycle = range(len(visited) + 1)
        selected = {}
        for x, y in edges:
            selected[x] = []
        for x, y in edges:
            selected[x].append(y)
        # print (selected)
        while unvisited:
            thiscycle = []
            neighbors = unvisited
            while neighbors:
                current = neighbors[0]
                thiscycle.append(current)
                unvisited.remove(current)
                neighbors = [j for j in selected[current] if j in unvisited]
            if len(cycle) > len(thiscycle):
                cycle = thiscycle
        return cycle
```

Time complexity of Min Max Algorithm: Time complexity of Min-Max algorithm is **$O(b^m)$**, where b is branching factor of the city-tree, and m is the maximum depth of the tree.

OUTPUT:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL
(190, [0, 2, 1, 3, 4, 0])
PS C:\Users\iQais>
```