**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](#_Toc60325160)

[**Breadth First Search and Depth First Search:** 4](#_Toc60325161)

[**Uniform Cost Search:** 8](#_Toc60325162)

[**A\* Search:** 9](#_Toc60325163)

[**Simulated Annealing on Tkinter:** 12](#_Toc60325164)

[**MIN MAX Algorithm:** 15](#_Toc60325165)

|  |
| --- |
| **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

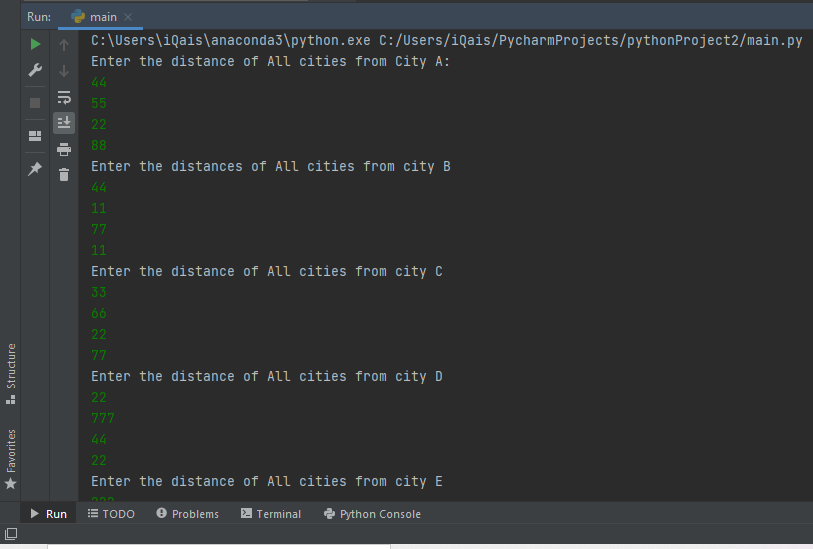
# **Breadth First Search and Depth First Search:**

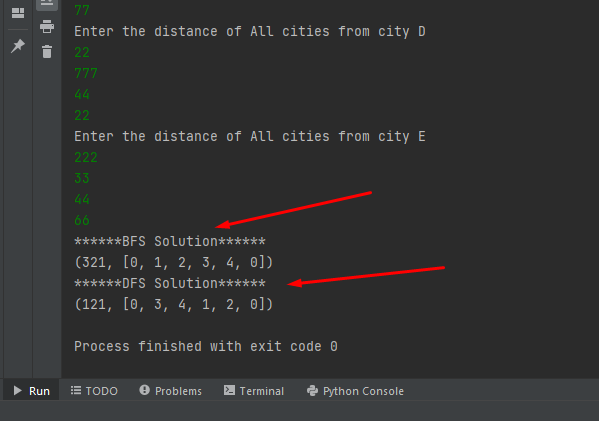
\_\_author\_\_ = "Qaiser Abbas"  
\_\_copyright\_\_ = "Copyright 2020, Artificial Intelligence Assignment-02"  
\_\_email\_\_ = "qaiserabbas889@yahoo.com"  
  
edges = []  
print("Enter the distance of All cities from City A:")  
edges\_cityA = []  
edges\_cityB = []  
edges\_cityC = []  
edges\_cityD = []  
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))  
  
  
  
  
  
  
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:**





# **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},

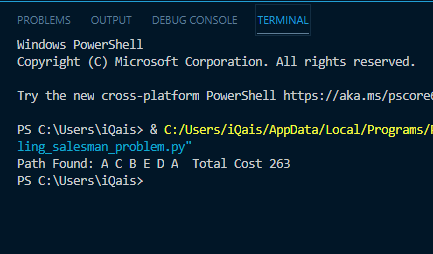
    'E': {'A': 190, 'B': 151, 'C': 199, 'D': 180},

    }

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\*/ε+1. Here we have taken +1, as we start from state 0 and end to C\*/ε. = **O(b1 + [C\*/ε])**

**OUTPUT:**

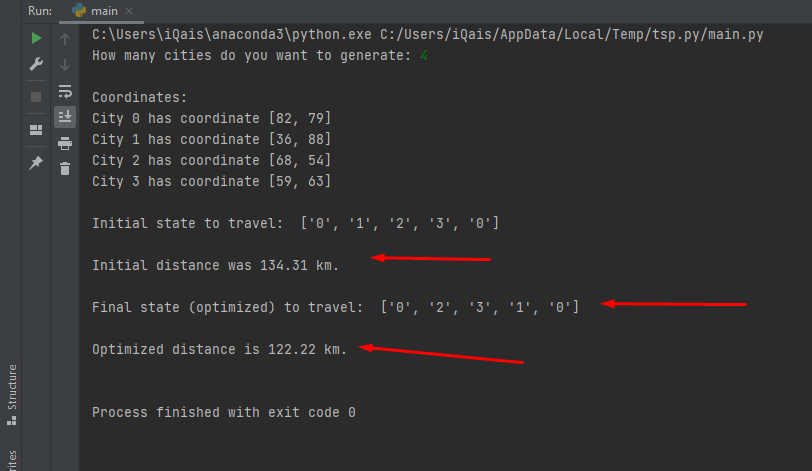


# **A\* Search:**

# 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:**



# **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

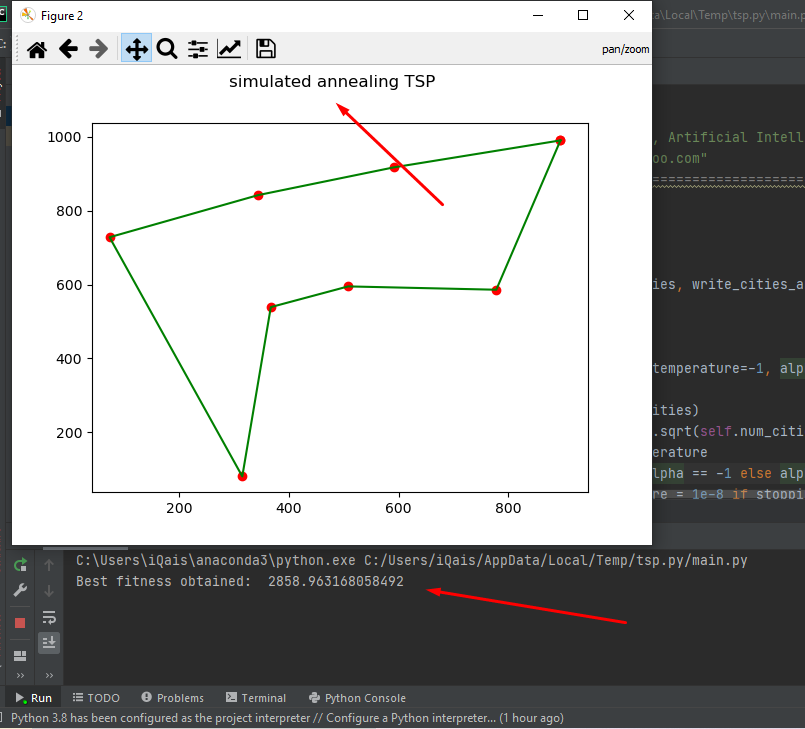
73 728

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.permutations(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(bm)**, where b is branching factor of the city-tree, and m is the maximum depth of the tree.

**OUTPUT:**

