Traveling salesperson report

1.

#!/usr/bin/python3  
from random import randrange  
  
from which\_pyqt import PYQT\_VER  
if PYQT\_VER == 'PYQT5':  
 from PyQt5.QtCore import QLineF, QPointF  
elif PYQT\_VER == 'PYQT4':  
 from PyQt4.QtCore import QLineF, QPointF  
elif PYQT\_VER == 'PYQT6':  
 from PyQt6.QtCore import QLineF, QPointF  
else:  
 raise Exception('Unsupported Version of PyQt: {}'.format(PYQT\_VER))  
  
  
  
  
import time  
import numpy as np  
from TSPClasses import \*  
from queue import PriorityQueue  
import copy  
import heapq  
import heapq  
import itertools  
import time  
  
  
  
  
class TSPSolver:  
 def \_\_init\_\_( self, gui\_view ):  
 self.\_scenario = None  
  
 def setupWithScenario( self, scenario ):  
 self.\_scenario = scenario  
  
  
 ''' <summary>  
 This is the entry point for the default solver  
 which just finds a valid random tour. Note this could be used to find your  
 initial BSSF.  
 </summary>  
 <returns>results dictionary for GUI that contains three ints: cost of solution,  
 time spent to find solution, number of permutations tried during search, the  
 solution found, and three null values for fields not used for this  
 algorithm</returns>  
 '''  
  
 def defaultRandomTour( self, time\_allowance=60.0 ):  
 results = {}  
 cities = self.\_scenario.getCities()  
 ncities = len(cities)  
 foundTour = False  
 count = 0  
 bssf = None  
 start\_time = time.time()  
 while not foundTour and time.time()-start\_time < time\_allowance:  
 # create a random permutation  
 perm = np.random.permutation( ncities )  
 route = []  
 # Now build the route using the random permutation  
 for i in range( ncities ):  
 route.append( cities[ perm[i] ] )  
 bssf = TSPSolution(route)  
 count += 1  
 if bssf.cost < np.inf:  
 # Found a valid route  
 foundTour = True  
 end\_time = time.time()  
 results['cost'] = bssf.cost if foundTour else math.inf  
 results['time'] = end\_time - start\_time  
 results['count'] = count  
 results['soln'] = bssf  
 results['max'] = None  
 results['total'] = None  
 results['pruned'] = None  
 return results  
  
  
 ''' <summary>  
 This is the entry point for the greedy solver, which you must implement for  
 the group project (but it is probably a good idea to just do it for the branch-and  
 bound project as a way to get your feet wet). Note this could be used to find your  
 initial BSSF.  
 </summary>  
 <returns>results dictionary for GUI that contains three ints: cost of best solution,  
 time spent to find best solution, total number of solutions found, the best  
 solution found, and three null values for fields not used for this  
 algorithm</returns>  
 '''  
  
 def greedy( self,time\_allowance=60.0 ):  
 pass  
  
  
  
 ''' <summary>  
 This is the entry point for the branch-and-bound algorithm that you will implement  
 </summary>  
 <returns>results dictionary for GUI that contains three ints: cost of best solution,  
 time spent to find best solution, total number solutions found during search (does  
 not include the initial BSSF), the best solution found, and three more ints:  
 max queue size, total number of states created, and number of pruned states.</returns>  
 '''  
  
 def branchAndBound( self, time\_allowance=60.0 ):  
 results = {}  
 cities = self.\_scenario.getCities()  
 num\_cities = len(cities)  
 matrix = np.empty((num\_cities,num\_cities))  
 children = []  
 rows\_not\_visited = []  
 columns\_not\_visited = []  
  
 # takes n^2 time to fill up the matrix with the cost of each city to another city  
 # takes n^2 space where n is number of cities  
 for i in range(num\_cities):  
  
 children.append(i)  
 rows\_not\_visited.append(i)  
 columns\_not\_visited.append(i)  
 for j in range(num\_cities):  
 matrix[i,j] = (cities[i].costTo(cities[j]))  
 starting\_city = randrange(num\_cities)  
 bssf = math.inf  
 nums\_visited = []  
 # The commented out code below was used for debugging  
 #m = [[math.inf,9,math.inf,8,math.inf],  
 # [math.inf,math.inf,4,math.inf,2],  
 # [math.inf,3,math.inf,4,math.inf],  
 # [math.inf,6,7,math.inf,12],  
 # [1,math.inf,math.inf,10,math.inf]]  
 path\_list = []  
 # creating a new node takes up O(n) time where n is the number of cities  
 # have to pass information from child to parent using deep copies making these deep  
 # copies takes O(n) time have to create new list copy all elements over to the new list  
 node = Node(path\_list,matrix,1,-1,starting\_city,nums\_visited,rows\_not\_visited,columns\_not\_visited,children)  
 #node.nums\_visited.add(node.city\_id)  
 node.cost = self.find\_cost(node)  
 nodes\_queue = []  
 # Every time we push a new node onto the queue it takes O(nlogn) time  
 # because have to sort all of the elements and push the min element to the top  
 heapq.heappush(nodes\_queue,(self.make\_key(node),node))  
 intermediate\_solutions = 0  
 total\_child\_state = 0  
 solution\_time = 0  
 num\_pruned = 0  
 route = []  
 max\_queue\_size = 0  
 start\_time = time.time()  
 # Will keep iterating until either the priority queue is empty or time runs out  
 # If time doesn't run out it will iterate for each node created that's added to the queue  
 # the time complexity will be O(SL) Where S is the number of nodes created  
 # and L is the time it takes to find the lower bound for each node created  
 # at lower levels in the tree L will be close to O(n^2) where n is number of cities  
 # at higher levels in the tree toward the bottom time complexity will be closer to O(n)  
 # the Space Complexity will be O(M) where M is the max size of the Queue  
 while nodes\_queue:  
 # This is a constant time operation  
 top = heapq.heappop(nodes\_queue)[1]  
 current\_time = time.time()  
 if current\_time - start\_time >= time\_allowance:  
 break  
 if top.cost > bssf:  
 num\_pruned += 1  
 continue  
  
 if top.current\_level == num\_cities:  
 top.path\_list.append((top.city\_id,starting\_city))  
 cost = top.cost  
 if cost < bssf:  
 bssf = cost  
 intermediate\_solutions += 1  
 route = top.path\_list  
 solution\_time = time.time() - start\_time  
 continue  
 else:  
 continue  
 # the number of children created varies  
 # at the first couple of levels the number of children is  
 # close to n  
 # at the last couple of levels the number of children generated is close to 0  
 # adding each child node takes O(n) time where n is the number because  
 # we have to copy information from parent node to child such as path, visited rows, visited columns etc.  
 # finding the cost of each child node takes at the worst case O(n^2) time  
 # as we get further down the tree finding the cost get's closer to O(n) time  
 # because we're only considering rows and columns not visited and as we  
 # go down the tree the number of rows and columns not visited keeps getting  
 # smaller and smaller  
 for j in range(1,len(top.children) + 1):  
 temp\_matrix = copy.deepcopy(top.matrix)  
 parent\_path\_list = copy.deepcopy(top.path\_list)  
 nums\_visited = copy.deepcopy(top.nums\_visited)  
 children = copy.deepcopy(top.children)  
 rows\_not\_visited = copy.deepcopy(top.rows\_not\_visited)  
 columns\_not\_visited = copy.deepcopy(top.columns\_not\_visited)  
 temp\_node = Node(parent\_path\_list,temp\_matrix,top.current\_level + 1,top.city\_id,top.children[j-1] + 1,nums\_visited,rows\_not\_visited,columns\_not\_visited,children)  
 total\_child\_state += 1  
 path\_cost = top.matrix[top.city\_id - 1][top.children[j - 1]]  
 temp\_node.cost = self.find\_cost(temp\_node) + top.cost + path\_cost  
 if temp\_node.cost != math.inf:  
 heapq.heappush(nodes\_queue,(self.make\_key(temp\_node),temp\_node))  
 else:  
 num\_pruned += 1  
 if len(nodes\_queue) > max\_queue\_size:  
 max\_queue\_size = len(nodes\_queue)  
  
 num\_pruned += len(nodes\_queue)  
 end\_time = time.time()  
 final\_route = []  
 # This take O(n) time were n is the number of cities  
 # have to loop through the route to find the associated cities  
 for i in range(len(route)):  
 if i == 0:  
 final\_route.append(cities[starting\_city - 1])  
 final\_route.append(cities[route[i][1] - 1])  
 else:  
 final\_route.append(cities[route[i][1] - 1])  
 found\_tour = False  
 if bssf != math.inf:  
 found\_tour = True  
 TSPSolver.\_bssf = TSPSolution(final\_route)  
 results['cost'] = bssf if found\_tour else math.inf  
 results['time'] = end\_time - start\_time  
 results['count'] = intermediate\_solutions  
 results['soln'] = TSPSolver.\_bssf  
 results['max'] = max\_queue\_size  
 results['total'] = total\_child\_state  
 results['pruned'] = num\_pruned  
 return results  
 print("Best search so far is " + str(bssf))  
 # parent\_path\_list contains the path of bssf  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
 pass  
  
  
  
 ''' <summary>  
 This is the entry point for the algorithm you'll write for your group project.  
 </summary>  
 <returns>results dictionary for GUI that contains three ints: cost of best solution,  
 time spent to find best solution, total number of solutions found during search, the  
 best solution found. You may use the other three field however you like.  
 algorithm</returns>  
 '''  
  
 def fancy( self,time\_allowance=60.0 ):  
 pass  
  
 def reduce\_row(self,node):  
 sum\_row = 0  
 # this will take at the worst case O(n^2) where n is the number of cities  
 # when we're at the beginning of the tree because we haven't visited any rows yet  
 for i in range(len(node.rows\_not\_visited)):  
 min = math.inf  
 for j in range(len(node.matrix[i])):  
 if node.matrix[node.rows\_not\_visited[i]][j] < min:  
 min = node.matrix[node.rows\_not\_visited[i]][j]  
 sum\_row += min  
 for j in range(len(node.matrix[i])):  
 node.matrix[node.rows\_not\_visited[i]][j] = node.matrix[node.rows\_not\_visited[i]][j] - min  
 return sum\_row  
  
 def reduce\_column(self,node):  
 sum\_column = 0  
 # this will take at the worst case O(n^2) where n is the number of cities  
 # when we're at the beginning of the tree because we haven't visited any columns yet  
 for i in range(len(node.columns\_not\_visited)):  
 min = math.inf  
 for j in range(len(node.matrix)):  
 if node.matrix[j][node.columns\_not\_visited[i]] < min:  
 min = node.matrix[j][node.columns\_not\_visited[i]]  
 sum\_column += min  
 for j in range(len(node.matrix)):  
 node.matrix[j][node.columns\_not\_visited[i]] = node.matrix[j][node.columns\_not\_visited[i]] - min  
 return sum\_column  
  
 def find\_cost(self,node):  
 row\_cost = self.reduce\_row(node)  
 column\_cost = self.reduce\_column(node)  
 return row\_cost + column\_cost  
  
 def make\_key(self,node):  
 return node.cost / node.current\_level  
  
  
  
  
  
  
  
  
class Node :  
 cost = 0  
 matrix = []  
 current\_level = 0  
 city\_id = 0  
 path\_list = []  
 nums\_visited = []  
 rows\_not\_visited = []  
 columns\_not\_visited = []  
 children = []  
  
 def \_\_init\_\_(self,path\_list,prev\_matrix,current\_level,i,j,nums\_visited,rows\_not\_visited,columns\_not\_visited,children):  
 self.current\_level = current\_level  
 self.city\_id = j  
 self.matrix = prev\_matrix  
 if current\_level == 1:  
 self.children = copy.deepcopy(children)  
 self.rows\_not\_visited = rows\_not\_visited  
 self.columns\_not\_visited = columns\_not\_visited  
 self.children.remove(j - 1)  
 if current\_level != 1:  
 self.path\_list = path\_list  
 self.path\_list.append((i,j))  
 self.nums\_visited = copy.deepcopy(nums\_visited)  
 self.children = copy.deepcopy(children)  
 #self.children.remove(j - 1)  
 self.rows\_not\_visited = copy.deepcopy(rows\_not\_visited)  
 self.columns\_not\_visited = copy.deepcopy(columns\_not\_visited)  
 #self.nums\_visited.add(j)  
 # len(prev\_matrix[i]) gives me the number of columns  
 # make every value in row I infinity  
 k = j  
 for f in range(len(self.matrix)):  
 self.matrix[i - 1][f] = math.inf  
 self.matrix[f][j-1] = math.inf  
 if f < len(self.children):  
 if self.children[f] == j - 1:  
 self.children.pop(f)  
 if f < len(self.rows\_not\_visited):  
 if self.rows\_not\_visited[f] == i -1:  
 self.rows\_not\_visited.pop(f)  
 if f < len(self.columns\_not\_visited):  
 if self.columns\_not\_visited[f] == j - 1:  
 self.columns\_not\_visited.pop(f)  
  
 #len(prev\_matrix) gives me the number of rows  
 # make every value in column J infinity  
  
 j = k  
  
 def \_\_lt\_\_(self, other):  
 return (self.current\_level < other.current\_level) and (self.cost < other.cost)

1. In the comments above
2. The data structure that I use to represent each state has a reduced cost matrix, has a city ID to keep track of which city it is, has a path list to keep track of how we got to that state, has a cost to know the cost of getting to that state, and also has rows and columns visited so I can reduce the time it takes to find the lower bound of each state as I get lower and lower in the tree.
3. I used a Heap which is from Pythons Heap library. It stores values as a binary tree where each parent is less than or equal to it’s children. If you insert a value it take logn time because it will there are logn levels in the tree and at each level it has to make a constant time comparison to make sure all of the elements are in the right order. This means that to get the element with the lowest value is an O(1) operation because the min element is always stored at the top of the tree so all you have to do is pull it off the top.
4. I decided to just set the initial BSSF to infinity, If the result is infinity this means that I never found a solution. I tried setting the initial bssf to the cost returned by the default tour so I could not waste time on any nodes where the cost of the node was greater than the bssf from the default tour but I didn’t notice a significant difference in speed so I decided to keep the initial bssf as math.inf

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Cities | Seed | Running Time (sec.) | Cost of the best tour found | Max # of stored states at a given time | # of BSSF updates | Total # of states created | Total # of states Pruned |
| 15 | 20 | 10 | 105340 | 93 | 19 | 12458 | 10668 |
| 16 | 902 | 13 | 7954 | 98 | 1 | 9993 | 8955 |
| 40 | 8 | 60 | 16918 | 13505 | 1 | 38862 | 32656 |
| 38 | 14 | 60 | 17496 | 930 | 10 | 35722 | 31356 |
| 31 | 25 | 60 | 13985 | 13308 | 5 | 62753 | 51970 |
| 33 | 22 | 60 | 15211 | 5831 | 5 | 51691 | 43465 |
| 12 | 10 | 9 | 8777 | 75 | 6 | 23202 | 18353 |
| 21 | 19 | 11 | 11905 | 204 | 3 | 12541 | 10880 |
| 24 | 23 | 20 | 12356 | 326 | 13 | 21415 | 18420 |
| 27 | 12 | 15 | 16095 | 4788 | 4 | 20464 | 15575 |

1. Table above
2. It takes a lot longer to find the optimal path for larger inputs because the tree has a lot of different paths to choose from going down so it has to generate a lot more child states and find the cost of each child state, the more child states generated means we’re going to prune more child states as evidenced by the rows above where the inputs where larger the more states where pruned. The max number of states stored at a time gets very big for larger inputs as well and when the time runs out we always prune every node that is left on the queue.
3. I found that the best approach for digging down in the tree as deep as possible early on was to make the key to the heap a combination of the cost and level. I decided to divide the cost by the level, this way if we had two nodes who’s costs were the same. We could divide by the level and the node with the higher level meaning the node deeper down in the tree would be chosen. This made it so we always chose the node with the lowest cost that was furthest down in the tree.