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
```

```python
    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_v
isited,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)
```

2. In the comments above

3. 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.

4. 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.

5. 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

6. Table above
7. 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.
8. 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.