Jonathan Armknecht Project 5 Report CS 312 Sec2

1.

```
#!/usr/bin/python3

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
else:
    raise Exception('Unsupported Version of PyQt: {}'.format(PYQT_VER))

import time
import numpy as np
from TSPClasses import *
import heapq
import copy
import itertools


class TSPSolver:
    def __init__(self, gui_view):
        self._scenario = None
        self.nodeQueue = []
        self.numSols = 0
        self.totalStates = 0
        self.bssf = None
        self.currState = None
        self.prunedStates = 0
        self.maxQSize = 0
        self.numStatesAddedToQ = 0
        self.cities = 0  # Stores the array of x y coords for each city
        self.cost = 0
        self.numBSSFUpdates = 0

    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>
    '''
    # O(n!) time since it runs till it finds a correct tour. This is because the worst case would be the number of all city
path permutations
    # which is O(n!). However the average case is more likely closer to O(n) because it most likely finds a random tour
without having to go
    # through all of the permutations.
    # The space complexity here is O(n) since we are storing a route of length n + 1 to
    # get back to the start city but the one is dropped.
    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:  # O(n!) this may have to
            # create a random permutation
            perm = np.random.permutation(ncities)
            route = []
            # Now build the route using the random permutation
            for i in range(ncities):  # O(n)
                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>
    '''
    # The O(n^3 *(n-1)!) is the worst case time complexity if no pruning happened. This is because the queue can have
up to
    # all the states on it except the first one made. However due to pruning this is not what the big O averages out to be
    # closer to O(n^3) because of my priority key making the search look at states that have the a lower cost and are
found deeper
    # within the tree. Thus the O(n^3) comes from O(n^2 * b^n) which since I travel down one state and try to get to the
bottom of the tree
    # right away becomes O(n^2 * n) since n the most children we can have on the first level and it goes down each level.
So the O(n^3) comes
    # from having to make all of the child nodes at the beginning cause there is less and less work going down the tree.
    # The space complexity is O(n^2) from making the 2d array
    def branchAndBound(self, time_allowance=60.0):
        results = {}  # O(1)
        self.cities = self._scenario.getCities()[:]  # Copies over x,y 2d array for each city so O(n^2)
        self.numSols = 0 # O(1)
        self.totalStates = 0 # O(1)
        self.prunedStates = 0 # O(1)
        self.nodeQueue = [] # O(1)
        self.maxQSize = 0 # O(1)
        self.numBSSFUpdates = 0 # O(1)
        self.numStatesAddedToQ = 0 # O(1)

        defaultResults = self.defaultRandomTour(time.clock())  # Grab the default route to use as initial BSSF O(n)
        self.bssf = defaultResults['soln']  # Take what is stored in the dictionary of defaultResults at soln key and use as
bssf
        print("Initial BSSF: ", self.bssf._costOfRoute()) # Checks that we got a BSSF
        start_time = time.time() # Record the start time so it can be compared with end time
        self.cost = self.bssf._costOfRoute()  # Stores initial cost of BSSF in cost
        city1 = copy.deepcopy(self.cities[0])  # Copy info of first city into city 1 O(n) deepcopy so it isn't a pointer

        city1.setCostMatrix(
          self.reduceMatrix(CostMatrix(self.cities[:])))  # O(n^2) since it copies n cities into a 2d array of size n

        # O(log n) is time complexity cause inserting a node into a binary tree can take at most the height of the tree
        # log n. Where n is the number of states pushed onto the heap.
        # Space complexity of the heap is O((n - 1)!) since that is the worst case size of the heap
        # this means that all of the states except the first city chosen are on the heap
        heapq.heappush(self.nodeQueue, city1)

        self.numStatesAddedToQ += 1  # State city1 was added to the queue so increment

        # This could at the worst case go for O(n^3 * (n - 1)!) where n is the number of states on the queue which could be
all of them
        #  but the first city state. However due to pruning on average it will be much better. Closer to O(n^3)
        while (len(self.nodeQueue) > 0) and ((time.time() - start_time) < time_allowance):
          self.maxQSize = max(len(self.nodeQueue), self.maxQSize) # O(1)
          self.lookAtState(heapq.heappop(self.nodeQueue))  # Takes about O(n^3) time
          self.maxQSize = max(len(self.nodeQueue), self.maxQSize)  # O(1)

        end_time = time.time() # O(1)
        results['cost'] = self.bssf._costOfRoute()  # O(1)
        results['time'] = end_time - start_time  # O(1)
        results['count'] = self.numSols  # O(1)
        results['soln'] = self.bssf  # O(1)
        results['max'] = self.maxQSize  # O(1)
        results['total'] = self.totalStates  # O(1)
        results['pruned'] = self.prunedStates  # O(1)
        return results

    ''' <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

    # As time goes on the time complexity of this averages out to be O(n^2) the space complexity here is O(1) since we
    # are only updating values
    def reduceMatrix(self, costMatrix):
        cost = costMatrix.getCost()  # O(1)
        numRows = len(self.cities)   # O(1)
        numCols = numRows  # O(1)
```

```
        # Reduces rows first worst case O(n^3) if have to go through and reduce each entry but some will
        # be inf or 0 as time goes on so it comes down to averaging to be about O(n^2)
        for row in range(0, numRows):  # O(n)
            minValue = math.inf  # Store what the min value is for each row  O(1)
            for col in range(0, numCols):  # O(n)
                matrixValue = (costMatrix.getMatrix())[col][row]  # O(1)
                minValue = min(minValue, matrixValue)  # Take the min of the two  O(1)

            if minValue > 0 and minValue < math.inf:  # Will be entered in less and less down the tree so it's for loop isn't
counted in overall time complexity
                cost += minValue  # Add minvalue to the total bound cost O(1)
                # Changes each entry in a column by the minValue found
                for col in range(0, numCols):  # O(n)
                    matrixValue = (costMatrix.getMatrix())[col][row]  # O(1)
                    costMatrix.setMatrixValue(col, row, matrixValue - minValue)  # O(1)

        # Reduces cols second wosrtcase O(n^3) if have to go through and reduce each entry but some will
        # be inf or 0 as time goes on so it comes down to averaging to be about O(n^2)
        for col in range(0, numCols):  # O(n)
            minValue = math.inf  # stores min value for each column O(1)
            for row in range(0, numRows):  # O(n)
                matrixValue = (costMatrix.getMatrix())[col][row]  # O(1)
                minValue = min(minValue, matrixValue)  # Take min of the two  # O(1)

            if minValue > 0 and minValue < math.inf:  # Will be entered in less and less down the tree so it's for loop isn't
counted in overall time complexity
                cost += minValue  # Add minvalue to the total bound cost O(1)
                for row in range(0, numRows):  # O(n)
                    matrixValue = (costMatrix.getMatrix())[col][row]  # O(1)
                    costMatrix.setMatrixValue(col, row, matrixValue - minValue)  # O(1)
        costMatrix.setCost(cost)  # O(1)
        return costMatrix

    # Overall the time complexity here is O(n^3) with space complexity of O(n^2) since it makes new child state matrix
    def lookAtState(self, currState):
        self.currState = currState  # O(1)

        # With the for loop being O(n) and creating a child being O(n^2) overall code here is O(n^3)
        if (self.contExploring()):  # Returns true if should still explore state's kids as time goes on will be true less and less
from the bound averaging to O(1)
            for childCity in range(0, len(self.cities)):  # O(n)
                if ((self.currState.getCostMatrix()).getMatrix())[childCity][self.currState.getIndex()] < math.inf:  # O(1) this
just checks that the matrix entry is less than inf
                    childState = self.makeChild(childCity)  # making a child is O(n^2) cause of 2d array same with space
                    self.totalStates += 1  # made a new state
                    if childState != None: # will be None if the cost of the child is greater than the BSSF so it is not always
entered cause of pruning
                        # O(log n) is time complexity cause inserting a node into a binary tree can take at most the height of the tree
                        # log n. Where n is the number of states pushed onto the heap.
                        # Space complexity of the heap is O((n - 1)!) since that is the worst case size of the heap
                        # this means that all of the states except the first city chosen are on the heap
                        heapq.heappush(self.nodeQueue, childState)

                        self.numStatesAddedToQ += 1  # inc num states added  # O(1)
                        self.maxQSize = max(len(self.nodeQueue), self.maxQSize) # O(1)

    # Overall time complexity averages to be O(1) since as time goes on more and more states will be pruned as BSSF is
updated
    # This makes it so the first if statement is only looked at. Space complexity is O(n^2) when we have to copy over the
2d array
    # however this also averages to O(1) for the same reason as time complexity
    def contExploring(self): # Returns true if the cost is less than BSSF otherwise it is pruned
    matrix = self.currState.getCostMatrix()  # O(1)
    currMatrix = self.reduceMatrix(matrix)  # O(n^2)

    self.currState.setCostMatrix(currMatrix)  # O(1)

    if currMatrix.getCost() > self.bssf._costOfRoute():  # O(1)
      self.prunedStates += 1  # O(1)
      return False  # Don't need to look at this states kids

    if len(self.currState.getPath()) == len(self.cities) + 1:  # we found a cycle and can update BSSF possibly O(1)
      path = self.currState.getPath()[:]  # O(n^2) for copying 2d array
      if (path[0].getName() == path[-1].getName()):  # the first and last city are the same and it is a cycle O(1)
        self.numSols += 1  # found a solution O(1)
        if (self.currState.getCostMatrix()).getCost() <= self.cost:  # O(1)
          self.numBSSFUpdates += 1  # O(1)
          self.bssf = TSPSolution(self.currState.getPath()[:-1])  # Get the path O(1)
          self.cost = (self.currState.getCostMatrix()).getCost()  # set the cost of the path O(1)
      return False  # found a leaf node
    return True

  # Overall time and space complexity are O(n^2) this comes from copying over the 2d array from the parent to the
child made
  def makeChild(self, childCity):
    childName = nameForInt(childCity + 1)  # O(1)
    currLevel = len(childName)  # O(1) this checks where in the tree we are based on the length of the name

    cost = (self.currState.getCostMatrix()).getCost() + ((self.currState.getCostMatrix()).getMatrix())[childCity][
      self.currState.getIndex()]  # O(1)

    if cost > self.bssf._costOfRoute():  # O(1)
      self.prunedStates += 1  # O(1)
      return None  # state was pruned and not needed checking this now saves overall space in the priority queue
    # O(1) by setting the key as the lower bound cost and the
    # level it is on ensures that smaller costs and those closer to the bottom have higher priority
    key = (cost / currLevel)

    childMatrix = CostMatrix(self.cities[:])  # O(n) copies over cities
    childMatrix.setMatrix(
      [row[:] for row in (self.currState.getCostMatrix()).getMatrix()])  # time and space both O(n^2) copying 2d array
    childMatrix.setCost(cost)  # O(1)
    for i in range(0, len(self.cities)):  # O(n) sets things to inf that need to be inf
      childMatrix.setMatrixValue(i, self.currState.getIndex(), math.inf)  # O(1)
      childMatrix.setMatrixValue(childCity, i, math.inf)  # O(1)
    childMatrix.setMatrixValue(self.currState.getIndex(), childCity, math.inf)  # O(1)
    # Copies parent(childCity) state's stuff into childNode O(n^2) for copying 2d array time and space complexity
    childNode = copy.deepcopy((self.cities[:])[childCity])

    childNode.setCostMatrix(childMatrix)  #O(1)
    childNode.setIndexAndName(childCity, childName)  #O(1)
    childNode.setKey(key)  # O(1)
    childNode.setPath(self.currState.getPath())  # O(1)
    childNode.addToPath(childNode)  # O(1)
    return childNode
#!/usr/bin/python3


import math
import numpy as np
import random
import time
```

```python
class TSPSolution:
  def __init__( self, listOfCities):
    self.route = listOfCities
    self.cost = self._costOfRoute()
    #print( [c._index for c in listOfCities] )

  def _costOfRoute( self ):
    cost = 0
    #print('cost = ',cost)
    last = self.route[0]
    for city in self.route[1:]:
      #print('cost increasing by {} for leg {} to {}'.format(last.costTo(city),last._name,city._name))
      cost += last.costTo(city)
      last = city
    #print('cost increasing by {} for leg {} to {}'.format(self.route[-1].costTo(self.route[0]),self.route[-
1]._name,self.route[0]._name))
    cost += self.route[-1].costTo( self.route[0] )
    #print('cost = ',cost)
    return cost

  def enumerateEdges( self ):
    elist = []
    c1 = self.route[0]
    for c2 in self.route[1:]:
      dist = c1.costTo( c2 )
      if dist == np.inf:
        return None
      elist.append( (c1, c2, int(math.ceil(dist))) )
      c1 = c2
    dist = self.route[-1].costTo( self.route[0] )
    if dist == np.inf:
      return None
    elist.append( (self.route[-1], self.route[0], int(math.ceil(dist))) )
    return elist


def nameForInt( num ):
  if num == 0:
    return ''
  elif num <= 26:
    return chr( ord('A')+num-1 )
  else:
    return nameForInt((num-1) // 26 ) + nameForInt((num-1)%26+1)







class Scenario:

  HARD_MODE_FRACTION_TO_REMOVE = 0.20 # Remove 20% of the edges

  def __init__( self, city_locations, difficulty, rand_seed ):
    self._difficulty = difficulty

    if difficulty == "Normal" or difficulty == "Hard":
      self._cities = [City( pt.x(), pt.y(), \
                 random.uniform(0.0,1.0) \
                        ) for pt in city_locations]
        elif difficulty == "Hard (Deterministic)":
            random.seed( rand_seed )
            self._cities = [City( pt.x(), pt.y(), \
                                  random.uniform(0.0,1.0) \
                        ) for pt in city_locations]
        else:
            self._cities = [City( pt.x(), pt.y() ) for pt in city_locations]


        num = 0
        for city in self._cities:
            #if difficulty == "Hard":
            city.setScenario(self)
            city.setIndexAndName( num, nameForInt( num+1 ) )
            city.addToPath(city)
            num += 1

        # Assume all edges exists except self-edges
        ncities = len(self._cities)
        self._edge_exists = ( np.ones((ncities,ncities)) - np.diag( np.ones((ncities)) ) ) > 0

        #print( self._edge_exists )
        if difficulty == "Hard":
            self.thinEdges()
        elif difficulty == "Hard (Deterministic)":
            self.thinEdges(deterministic=True)

    def getCities( self ):
        return self._cities


    def randperm( self, n ):            #isn't there a numpy function that does this and even gets called in Solver?
        perm = np.arange(n)
        for i in range(n):
            randind = random.randint(i,n-1)
            save = perm[i]
            perm[i] = perm[randind]
            perm[randind] = save
        return perm

    def thinEdges( self, deterministic=False ):
        ncities = len(self._cities)
        edge_count = ncities*(ncities-1) # can't have self-edge
        num_to_remove = np.floor(self.HARD_MODE_FRACTION_TO_REMOVE*edge_count)

        #edge_exists = ( np.ones((ncities,ncities)) - np.diag( np.ones((ncities)) ) ) > 0
        can_delete = self._edge_exists.copy()

        # Set aside a route to ensure at least one tour exists
        route_keep = np.random.permutation( ncities )
        if deterministic:
            route_keep = self.randperm( ncities )
        for i in range(ncities):
            can_delete[route_keep[i],route_keep[(i+1)%ncities]] = False

        # Now remove edges until
        while num_to_remove > 0:
            if deterministic:
                src = random.randint(0,ncities-1)
                dst = random.randint(0,ncities-1)
            else:
```

```
            src = np.random.randint(ncities)
            dst = np.random.randint(ncities)
        if self._edge_exists[src,dst] and can_delete[src,dst]:
            self._edge_exists[src,dst] = False
            num_to_remove -= 1

    #print( self._edge_exists )




class City:
  def __init__( self, x, y, elevation=0.0 ):
    self._x = x
    self._y = y
    self._elevation = elevation
    self._scenario = None
    self._index = -1
    self._name = None
    self.costMatrix = None
    self.key = None
    self.path = []

  def __lt__(self, other):  # Compares priorities to see where it belongs in queue
    if self.key < other.getKey():  # O(1)
      return 1
    elif other.getKey() < self.key:  # O(1)
      return -1
    else:
      return 0  #  Same priority

  def __cmp__(self, other):  # See if the city is the same through the name
    if self._name == other.getName(): # O(1)
      return True
    else:
      return False

  def addToPath(self, cityToAdd):  # Holds the cities already visited
    self.path.append(cityToAdd)  # O(1)

  def setPath(self, path):
    self.path = path[:]  #  O(1)Sets the states path

  def getPath(self):
    return self.path  # O(1)

  def setKey(self, key):
    self.key = key  # O(1)

  def getKey(self):
    return self.key  # O(1)

  def setCostMatrix(self, costMatrix):
    self.costMatrix = costMatrix  # O(1)

  def getCostMatrix(self):
    return self.costMatrix  # O(1)

  def getName(self):
    return self._name  # O(1)

  def getIndex(self):
    return self._index  # O(1)

  def setIndexAndName( self, index, name ):
    self._index = index  # O(1)
    self._name = name  # O(1)

  def setScenario( self, scenario ):
    self._scenario = scenario  # O(1)

  ''' <summary>
    How much does it cost to get from this city to the destination?
    Note that this is an asymmetric cost function.

    In advanced mode, it returns infinity when there is no connection.
    </summary> '''
  MAP_SCALE = 1000.0
  def costTo( self, other_city ):

    assert( type(other_city) == City )

    # In hard mode, remove edges; this slows down the calculation...
    # Use this in all difficulties, it ensures INF for self-edge
    if not self._scenario._edge_exists[self._index, other_city._index]:
      #print( 'Edge ({},{}) doesn\'t exist'.format(self._index,other_city._index) )
      return np.inf

    # Euclidean Distance
    cost = math.sqrt( (other_city._x - self._x)**2 +
                      (other_city._y - self._y)**2 )

    # For Medium and Hard modes, add in an asymmetric cost (in easy mode it is zero).
    if not self._scenario._difficulty == 'Easy':
      cost += (other_city._elevation - self._elevation)
      if cost < 0.0:
            cost = 0.0
    #cost *= SCALE_FACTOR


    return int(math.ceil(cost * self.MAP_SCALE))

class CostMatrix:
  def __init__(self, cityList):
    self.matrix = [[math.inf] * len(cityList) for i in range(len(cityList))]  # Matrix of the state O(n^2)
    self.cost = 0  # bound of the matrix O(1)

    # O(n^2) for setting up the cost matrix n is the number of cities
    for row in range(0, len(cityList)):
      for col in range(0, len(cityList)):
            self.matrix[col][row] = cityList[row].costTo(cityList[col])

  def getMatrix(self):
    return self.matrix # O(1)

  def setMatrix(self, matrix):
    self.matrix = matrix # O(1)

  def setMatrixValue(self, x, y, value):
    self.matrix[x][y] = value # O(1)

  def getCost(self):
    return self.cost # O(1)
```

```
def setCost(self, cost):
    self.cost = cost # O(1)
```

2.

makeChild Function: This function is where I make more search states that are either pruned right away or not. This is where I make my priority key for the state for when it is put into the priority queue is push. It gets the level of the state by looking at its name. This length of the name tells us how far down the tree we are with longer city names being at the bottom of the tree. This is how I give priority to those states that are closer to solving the route. Because if the length of the name is longer than it makes the overall key smaller since it divides the cost by the level. A lower key was given higher priority. If the cost of the current search state was greater than the BSSF it was pruned which happened in constant time. However it if was lower I then made the new reduced matrix that was based off of the parent's reduced matrix and set that search states' cost, key, path, and index and name. Since I was copying over the matrix and setting each entry to the correct number the **overall time and space complexity are O(n^2) this comes from copying over the 2d array from the parent to the child made.**

contExploring Function: This function was a helper function to lookAtState. It would help determine if the search state needed to be expanded into other states. This function would look at if the cost of the current states matrix was greater than the BSSF cost. If it was it would prune the state. Meaning that it would throw the state away after popping it off of the queue and not even look at it. If it wasn't greater than the BSSF I checked to see if the path the state stores has the same first and last city in it. If it does I have found a leaf node and I then I check the current lowest cost and compare it to the cost from the state's matrix. If it is lower I update BSSF. Time and space complexity is O(n^2) HOWEVER as time goes on more and more states will be pruned as the BSSF is updated and gets smaller. This makes it so that the first if statement is only looked at. Making on **average a Big-O closer to that of O(1) as time goes on.**

lookAtState Function: This function calls contExploring to see if the search state needs to be further explored. If it does then it goes through all of the cities which is O(n). Then it looks at each the matrix where the city and the currState is and checks to see if it is less than infinity. If it is then that means we can reach the city from the currState and thus we need to make a new state by calling makeChild. Making a child is O(n^2) as seen above. It then checks to see if then if the child was pruned by seeing if it was None. If it is not none then it adds the state to the heap with a push which is O(log n). HOWEVER on as time goes on more and more states are pruned as our BSSF gets lower and lower so None will be returned more and more. **Thus on average the overall time complexity here is O(n^3).** This comes from the loop that goes through all the cities and making the new child state. **The space complexity here is O(n^2) since it make a new child state matrix.**

reduceMatrix Function: I decided to reduce rows first and then columns. This gets the number of cities so as to check to see how many rows and columns are in the

matrix of the current state. When reducing rows I set the minValue to infinity. Then I go through all of the columns. I get the value at that column and row and compare it to the minValue and take which ever is the smallest. If the minValue was not 0 or infinity then I update the cost of that state by adding the minValue to it and then changing all of the matrixValue by substrating the minValue from them. These both happen in constant time. HOWEVER, this if minValue > 0 and minValue < math.inf will be entered less and less down the tree because all of the entries will either become infinity or 0. Thus on average this loop's time complexity isn't really counted. So row reducing on average since we won't have to reduce every row as time goes on row reducing comes down to being O(n^2) on average.
Next for reducing columns I do the same thing as with reducing rows but this time I'm reducing columns. Matrix values are updated the same way as before and happens in constant time. As with rows time complexity, time complexity with column reducing is also O(n^2) for the same reason. As time goes on there will be less columns to reduce because more of them will have 0 in its column or the whole column is infinity. **Thus the overall average time complexity is O(n^2) and the space complexity here is O(1) since we are only updating values.**

<u>branchAndBound Function:</u> This function starts off by calling defaultRandomTour to initiate my BSSF. The defaultRandomTour runs on average O(n) time. Worst case for the defautRandomTour is O(n!). This is because the worst case to find a tour would be that it would have to go through all of the permutations which is O(n!). However the average case is more closer to O(n) because it most likely will find a random tour without having to go through all of the permutations. The space complexity here is O(n) since we are storing a route of length n + 1 to get back to the start city but the one is dropped. Then it grabs the first city in the list of cities and sets this as the city1. It then sets up the reduced matrix for this state by giving the reduceMatrix function the matrix of all the cities. This happens in O(n^2) time and space. Then it pushes this onto my heap. Pushing and poping from my heap is O(log n). It is O(log n) time complexity cause inserting a node into a binary tree can take at most the height of the tree log n. Where n is the number of states pushed onto the heap. The space complexity of the heap is O((n - 1)!) since that is the worst case size of the heap this means that all of the states except the first city chosen are on the heap all at once.
Then I enter a while loop until the queue is empty or the time allowance is done. This loop updates the maxQSize if it has changed. Calls lookAtState giving it the popped off node at the top of the queue. Popping takes O(log n) but lookAtState runs in O(n^3) time so this dominates. Therefor this while loop could at the worst case go for O(n^3 * (n - 1)!) where n is the number of states on the queue which could be all of them but the first city state. However due to pruning on average it will be much better. On average it will be closer to O(n^3).

O(n^3 *(n-1)!) is the worst case time complexity of my branch and bound algorithm if no pruning happened. This is because the queue can have up to all the states on it except the first one made. However due to pruning this is not what the big O is. It averages out to be closer to O(n^3) because of my priority key making the search look at states that have a lower cost and are found deeper within the tree. Thus the O(n^3) comes from O(n^2 * b^n) which since I travel down one state and try to get to the bottom of the tree right away becomes O(n^2 * n) since n is the most children we can have on the first level and it goes down each level. So the O(n^3) comes from having to make all of the child nodes at the beginning cause there is less and less work going down the tree. The space complexity is O(n^2) from making the 2d array matrix for each state.

Thus the overall time complexity for my Branch and Bound algorithm is O(n^3) with a space complexity of O(n^2).

3.
<u>Data Structures used to represent the search states:</u>
Each state had a 2d array used as that states cost matrix. Each state also had an index and a name of the state, cost/lower bound, and a path of visited cities. These were all assigned in the makeChild function which is explained above. These were stored in a class called CostMatrix . Each city had one of these objects, which made it easier to set the key, path, cost, matrix as well as retrieve each one of this variables. Each city's path contained where it had come from as well as which current city it was in.

4.
<u>Priority Queue Data Structure Used:</u>
I used the built in Python priority queue heapq. I was able to set my key as the lower bound (cost) divided by the current level it was on. The further down the tree the higher the level number which when using this number to divide the lower bound the key gets smaller which gives that state a higher priority. This makes sure that both bound and tree depth are looked at when deciding what state to explore next. See makeChild function for more information.

5.
<u>Initial BSSF Approach:</u>
I decided to save time and not implement the greedy algorithm. I figured I would concentrate on this once I started the group project with my group. I did this to save time so I could put my full concentration on getting the branch and bound algorithm working. So what I did instead to get my initial BSSF is that I called the defaultRandomTour once to get a BSSF that was an actual tour and wasn't a cost of infinity. I could have called it more than once and then took the lowest BSSF that it gave back so my pruning could have started earlier but I felt like that was unnecessary as my pruning still eventually started to prune.

6.

# Cities	Seed	Running time (sec.)	Cost of best tour found (*=optimal)	Max # of stored	# of BSSF updates	Total # of states	Total # of states

				states at a given time		created	pruned
15	20	19.1355	9394*	1230	56	57208	46071
16	902	37.4805	8305*	4603	43	87193	71187
23	1023	60.0026	20444	17993	62	62313	31098
27	40	60.0027	32001	24039	13	43575	12853
50	17	60.0097	71248	11593	2	12578	117
10	5	0.20742	9486*	102	10	670	464
19	169	60.0003	12311	13152	39	117595	85094
22	33	60.0047	18927	16096	29	74899	42955
17	799	52.8700	9246*	4072	81	154151	129612
15	918	6.91101	10716*	1090	17	17856	14720

7.
<u>Table Analysis:</u>

As cities were added the amount of time for it to run and find the best path increased. It increased factorially since searching for the best route is at the worst case O(n!). Just adding one city as seen when going from 15 to 16 just about doubles the time it takes to solve the problem. With the way I implemented the branch and bound algorithm when one searches 18 (not shown) or more cities the algorithm times out and just returns the BSSF. Thus my version can find the optimum tour with 17 or less cities.

The number of states created generally increased with some exceptions. These exceptions came about because of how many updates to BSSF. If BSSF was updated less then less states were pruned and thus my algorithm got stuck at looking at states that were already created trying to get to a leaf node that was a better cost than the BSSF which it never found. Since the algorithm had to dig deeper into the tree with these exceptions, new states were not found often before time expired. If it was ran with a higher time limit then more states would have been created.

There is a pattern seen with the number of states created and those that were pruned. When taking the ratio of (pruned states / created states) * 100 it was seen that: 10 cities = ~70%, 15 cities = ~80%, 16 cities = ~82%, 17 cities = ~84%, 19 cities = ~72%, 22 cities = ~57%, 23 cities = ~50%, 27 cities = ~29%, and 50 cities = ~ .9%. Thus when my algorithm was about to solve the problem in less than 60 seconds the percentage of pruned states increased with an increase in number of cities, and when it wasn't able to solve it in 60 sec the percentage of pruned states decreased with an increase in number of cities. This is because of the number of times BSSF is updated. The trend is that as more cities are added to the overall possible tour then the amount of times BSSF is updated is lower. Since it is lower there is less and less pruning because the algorithm gets stuck trying to get to a leaf node so it can update the BSSF for more pruning. Since there is less and less pruning there are less states created since we are stuck on the same level of the tree instead of going further down the tree cause all of the states at a level have the same or close to the same priority. Thus fewer states were created. However if time was unlimited I would expect to see the ratio keep on increases with the number of states pruned and the number of states created increasing as well for those that weren't solved with the optimal tour.