Trevor Ashby

Travelling Salesperson

1. Code

** See Appendix for Code

2. Time and Space Complexity

Priority Queue: sort, insert, pop.
Time: O(nlogn) Space: O(n^2 n!)
My Priority Queue takes O(nlogn) time because both insertion and popping are constant time. However, when I'm sorting my list, this will take O(nlogn) time because python uses TimSort which is a hybrid of insertion sort and merge sort.

Search States: tuples
Time: O(1) Space: O(n^2)
I use tuples, which do not take any time to manipulate. They require O(n^2) space however to store the matrix for the related decision/state.

Reduce Cost Matrix and updating it: reduce function.
Time: O(n^2) Space: O(n^2)
This takes O(n^2) time because when checking the row/column lowest value to update and checking to see if each row has a 0. Afterwards checking each column for 0's.

BSSF Initialization: Greedy.
Time: O(n^3) Space: O(n^2)
Started with an initial Greedy Algorithm which takes O(n^3). The reason for it being n^3 is I choose the best Greedy BSSF using each city as a starting city. This helps the Branch and Bound algorithm when deciding to trim or not trim a solution. Therefore, decreasing the chances of O(n!) time.

Expanding one Search State into others:
This will be constant time and space because upon expanding, a new search state will be created. Therefore, see above "Search States" for Time and Space.

The full Branch and Bound algorithm:
Time: O(n^2 n!) Space O(n^2 n!)
The full Branch and Bound algorithm will take O(n^2 n!) time and space because worst case scenario, the while loop which empties the priority queue will run for n! time. This can be shortened due to pruning, however at most it will run n! times. Then for each run of the while loop. The matrix will have to be reduced based upon the decision. Each reduction has O(n^2) time complexity. Therefore, in all this portion will take O(n^2 n!) time. Other actions occur in the Branch and Bound, greedy for example. But due to the max rule, these are negligible.

3. State Data Structures

I represented each state as a tuple that was inserted into the priority queue. Each tuple contained the matrix related to the decision, the route so far, the route names so far, the score of decision, the column of chosen destination (this is used for future computations once expanded), the level (how deep the decision is in the tree), calculated score based on lowerBound – (level * offset).

4. Priority Queue Data Structure

For my priority queue, I inserted tuples (as described above in #3) which contained the needed data for each state. The priority queue was sorted in descending order based off a "score" that was assigned to each state. This score was calculated by subtracting from the lowerBound of the state by the level/deepness of the state multiplied by a constant level multiplier. The reason for this was to prioritize deeper states that were closer to a solution rather than states near the top of the tree which needed many more computations to find an answer. This helped with speed improvements and finding a BSSF better than the initial BSSF in better time. Each iteration I would pop() from the Priority Queue and use that tuple as the current decision to be expanded. If the expanded decision deserved to be inserted into the queue, it was simply appended. Sorting would occur at each iteration to make sure that the queue was in the proper order so the right state would become expanded.

5. Approach for initial BSSF

For my initial BSSF I used a form of greedy algorithm to calculate a BSSF which was used for initial trimming and calculation. This greedy algorithm would start at the first node and calculate a greedy solution, then continue that process calculating greedy solutions from each node in the list. Finally, the lowest of all these solutions is returned. This allows for less error to occur during the "Hard" difficulty.

6. Table

# Cities	Seed	Running time (sec.)	Cost of best tour found (*=optimal)	Max # of stored states at a given time	# of BSSF updates	Total # of states created	Total # of states pruned
15	20	0.18309	8781	76	1	3330	3108
16	902	2.588	8059	74	2	36400	34124
17	968	27.204	10192	417	7	295936	278522
15	947	1.002	7891	33	0	15915	14855
16	120	8.028	9207	150	0	99872	93631
18	542	60	10684	838	7	786366	742673
20	295	60	11081	277	4	551380	523808
22	626	60	12462	373	9	556116	530830
26	829	60	13234	970	0	491166	472276
50	363	60	23133	2031	2	229150	224566

7. Results of Table

A couple of observations I made while gathering the data for the table above:

- The more cities that had infinite routes to them, the more likely it was that the original greedy BSSF was the most optimal one.
- As problem size increased, the total number of states increased similarly the number of pruned states would increase. This was expected. The number of infinite distanced cities also contributed to this.
- It seemed that the moment I went above 17 cities that the time taken jumped up to 60 seconds for most every case. I think this has to do with my LVL_MULTIPLIER variable. I think that if I increased this offset that the threshold where solutions took the 60 seconds would increase.
- I noticed that the max # of stored states seems to also be correlated to # of BSSF updates as well. An odd case is for 26 cities. There were a lot of max states, however no better solution was found. (I think that the infinite distance cities could have contributed to this).

8. Mechanisms for finding solutions early and to dig deeper

As mentioned above in #4, I used a form of “score” for each state that was calculated using the equation: ‘lowerBound – (lvl * LVL_MULTIPLIER)’. My Priority Queue was sorted based upon this calculated score. This caused my queue to prioritize states that were “deeper” and were closer to finding solutions rather than states that had just begun, which would require much more calculation to find an answer. I adjusted this LVL_MULTIPLIER until I found a balance and solution finding time that I liked.

Appendix

```
#!/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 itertools



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>
'''

#########################
# Time Complexity O(n^3)
# Space Complexity O(n^2)
#########################
def greedy( self,time_allowance=60.0 ):
    # initialize results, cities, ncities, and other variables
    results = {}
    cities = self._scenario.getCities()
    count = 0

    start_time = time.time()
    lowest_bssf = None
    lowest_bssf_cost = math.inf
    bssf = None
    foundTour = False

    # run a loop that will go through each city as the starting point
    for pos in range(len(cities)):
        startingCity = cities[pos]
        edges = startingCity._scenario._edge_exists
        curCity = startingCity
        route = []
        routeNames = []
        route.append(startingCity)
        routeNames.append(startingCity._name)
        for i in range(len(cities)):
            minLength = math.inf
            minName = "blank"
            minIndex = .5
            for edge_pos in range(len(cities)):
                dist = curCity.costTo(cities[edge_pos])
                if dist < minLength and cities[edge_pos]._name not in routeNames:
                    minLength = dist
                    minName = cities[edge_pos]._name
                    minIndex = cities[edge_pos]._index
            if minName != "blank":
                routeNames.append(cities[minIndex]._name)
                route.append(cities[minIndex])
                curCity = cities[minIndex]
        bssf = TSPSolution(route)
        count += 1
```

```
            if bssf.cost < lowest_bssf_cost:
                lowest_bssf = bssf
                lowest_bssf_cost = lowest_bssf.cost
        if lowest_bssf.cost < np.inf:
            # Found a valid route
            foundTour = True
        end_time = time.time()
        results['cost'] = lowest_bssf.cost if foundTour else math.inf
        results['time'] = end_time - start_time
        results['count'] = count
        results['soln'] = lowest_bssf
        results['max'] = None
        results['total'] = None
        results['pruned'] = None
        return results
        # pass

    ########################
    # Time Complexity O(n^2)
    # Space Complexity O(n^2)
    ########################
    def reduce(self, costMatrix, numCities, lowerBound):
        # create initial reduced cost matrix & initial lower bound
        # make sure that every row has a 0
        for row in range(numCities):
            # go through each column and find lowest value or 0
            hasZero = False
            lowestVal = math.inf
            for col in range(numCities):
                if lowestVal > costMatrix[row][col]:
                    lowestVal = costMatrix[row][col]
                if costMatrix[row][col] == 0:
                    hasZero = True
            if hasZero:
                continue
            else:
                if lowestVal != math.inf:
                    lowerBound += lowestVal
                    for col in range(numCities):
                        costMatrix[row][col] -= lowestVal

        # make sure that every col has a 0
        for col in range(numCities):
            hasZero = False
            lowestVal = math.inf
            for row in range(numCities):
                if lowestVal > costMatrix[row][col]:
                    lowestVal = costMatrix[row][col]
                if costMatrix[row][col] == 0:
                    hasZero = True
            if hasZero:
                continue
            else:
                if lowestVal != math.inf:
                    lowerBound += lowestVal
                    for row in range(numCities):
                        costMatrix[row][col] -= lowestVal

        return costMatrix, lowerBound

    def infiniteRowsAndColumns(self, inputMatrix, rowIn, colIn):
        returnMatrix = inputMatrix.copy()
        for row in range(len(returnMatrix)):
            for col in range(len(returnMatrix)):
                if col == colIn or row == rowIn:
                    returnMatrix[row][col] = math.inf
        return returnMatrix

    ''' <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 ):
        LVL_OFFSET = 450
        # get the first greedy bssf that can be used for initial trimming
        greedyResult = self.greedy()
        ##########################
        # Time Complexity O(n^3)
        # Space Complexity O(n^2)
        ##########################
        bssf = greedyResult['soln']

        # initialize results, cities, ncities, and other variables
        results = {}
        cities = self._scenario.getCities()
        numCities = len(cities)
        count = 0
        start_time = time.time()

        # generate unreduced cost matrix
        costMatrix = np.full((numCities,numCities), np.inf)
        # loop through each row filling in matrix
        for row in range(numCities):
            ##########################
            # Time Complexity O(n^2)
            # Space Complexity O(n^2)
            ##########################
            # loop through each column for every row filling in cost
            for col in range(numCities):
                costMatrix[row][col] = cities[row].costTo(cities[col])

        # create initial reduced cost matrix & initial lower bound
        totalStates = 0
        numTrimmed = 0
        lowerBound = 0
        foundTour = False
        ##########################
        # Time Complexity O(n^2)
        # Space Complexity O(n^2)
        ##########################
        costMatrix, lowerBound = self.reduce(costMatrix, numCities, lowerBound)

        # create priority queue
        priorityQueue = []
        route = []
        routeNames = []

        # GLOBAL CONST LVL_OFFSET 190
        # score - (lvl * lvl_offset)

        route.append(cities[0])
        routeNames.append(cities[0]._name)
        priorityQueue.append((costMatrix, route, routeNames, lowerBound, 0, 1, lowerBound - (1 *
LVL_OFFSET)))
        queueMaxSize = len(priorityQueue)
        # push 'A' onto the Queue, and calculate from there
        # calculate potential paths, start evaluating paths
        ##########################
        # Time Complexity O(n^2 * b^n)
        # Space Complexity O(n^2 * b^n)
        # Worst Case Scenario O(n^2 n!)
        ##########################
        while priorityQueue and time.time() - start_time < time_allowance:
            if len(priorityQueue) > queueMaxSize:
                queueMaxSize = len(priorityQueue)

            priorityQueue.sort(key=lambda tup: tup[6], reverse=True)
```

```
            currentDecision = priorityQueue.pop()
            if len(currentDecision[1]) == numCities and currentDecision[3] < bssf.cost:
               count += 1
               bssf = TSPSolution(currentDecision[1])
               foundTour = True
            else:
               for col in range(numCities):
                  totalStates += 1
                  tempMatrix = currentDecision[0].copy()
                  tempScore = currentDecision[3]
                  tempRoute = currentDecision[1].copy()
                  tempRouteNames = currentDecision[2].copy()
                  tempMatrix[col][currentDecision[4]] = math.inf
                  tempScore += tempMatrix[currentDecision[4]][col]
                  if tempScore > bssf.cost:
                     numTrimmed += 1
                     continue
                  else:
                     if cities[col]._name not in tempRouteNames:
                        tempRoute.append(cities[col])
                        tempRouteNames.append(cities[col]._name)
                     tempMatrix = self.infiniteRowsAndColumns(tempMatrix, currentDecision[4], col)
                     tempMatrix, tempScore = self.reduce(tempMatrix, numCities, tempScore)
                     if tempScore > bssf.cost:
                        numTrimmed += 1
                        continue
                     else:
                        priorityQueue.append((tempMatrix, tempRoute, tempRouteNames, tempScore, col,
currentDecision[5] + 1, tempScore - ((currentDecision[5] + 1) * LVL_OFFSET))) # TAKE INTO CONSIDERATION
DEEPNESS
         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'] = queueMaxSize
         results['total'] = totalStates
         results['pruned'] = numTrimmed + len(priorityQueue)
         return results
         # 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
```