Caleb Johnson
CS 312

Project 5
Traveling Salesperson

---

1. Code included at the end to improve readability

---

2. Complexities

Greedy Algorithm

Overall Time Complexity: O(n^2)
Overall Space Complexity: O(n)
BSSF Initialization
Time: Constant
Space: O(n) - stores each city

Branch-and-Bound Algorithm

Overall Time Complexity: O(2^n *n^2) for worst case, but closer to O(n^2 * k), where k is the total number of states checked, on average
Overall Space Complexity: O(n^2)
Priority Queue - I used the provided heapq Python function
Time: O(log(n)) for both heappush and heappop as it reheapifies cities
Space: O(n)
BSSF Initialization - This is equivalent to the greedy algorithm's runtime.
Time: O(n^2)
Space: O(n)
Reduced Cost Matrix
Time: O(n^2)
Space: O(n^2)

---

3. Data Structures for States

I utilized an increasingly incremented integer value to help me keep a tally of all created states as I went, with a separate variable that I also incremented for pruned states. I used a reduced matrix (implemented using numpy) to explore relative costs and see if there are more nodes to explore/prune. This took (n^2) time and space.

4. Priority Queue Data Structure

I used the provided heapq package for my priority queue instantiation. Heapq is a binary heap package that allows us to use push and pop functions that will re-heapify to provide us with priority queue capabilities.

---

5. Initial BSSF

For my branch-and-bound algorithm, I used the result of my greedy algorithm to select my initial best solution so far. For relatively small city numbers, our greedy algorithm will run very quickly, and provides us a much better starting point than a random solution might.

---

6. Results Table

Scenario			**Greedy**		**Branch-and-Bound**					
Numher of Cities	*Seed*	*Difficulty*	*Time*	*Length*	*Runtime*	*Cost Best Tour *Optimal*	*Max # Stored States*	*# BSSF Updates*	*# Total Staes*	*# Total Pruned States*
15	20	Hard	0.007	11,081	10.80	9,836*	44	21	15,785	12,829
16	902	Hard	0.007	11,654	58.23	8,051*	60	48	73,557	61,450
18	34	Hard	0.011	10,992	60	9,774	86	8	69,203	56,422
17	1	Hard	0.006	11,747	60	11,636	47	3	76,234	64,979
22	1410	Hard	0.016	13,106	60	15,181	97	0	60,546	52,754
11	1969	Hard	0.004	8,920	1.27	9,110*	19	14	869	638
14	24	Hard	0.007	9,998	6.69	8,716*	31	10	9,922	8,270
18	8	Hard	0.008	12,833	60	12,430	83	6	72,807	60,142
19	19	Hard	0.008	10,730	60	10,301	77	7	67,339	57,674
12	9	Hard	0.002	11,290	2.23	8,649*	33	29	3,660	2,766

---

7. Analysis of Report Table

We seem to observe a factorial increase in states as we increase our city allotment, which is understandable with our worst case time complexity. Total states increase drastically as we increase the city allotment as is expected. We are able to find different solutions and update our BSSF with small city sizes, and on some large counties we were not able to update at all.

One fascinating take-away for me is still the overall variability in the data based on the number of cities. This is likely just due to random chance and the fact that our initial state provided by our greedy algorithm is so important, but it is still surprising to see relatively small total states for our 22-city scenario compared to other lesser city count scenarios.

```python
#!/usr/bin/python3

"""
    Caleb Johnson
    CS 312 - Winter 2020
    April 6, 2020
"""

from PyQt5.QtCore import QLineF, QPointF



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

from copy import copy, deepcopy

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
```

```python
            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>
    '''

    """
        This is a greedy algorithmic approach to finding the solution to the TSP.


        Time Complexity: O(n^2) - We go through each node and also check all the neightbors of the
                                node to choose our best cost next city. This is O(n^2) and is not optimal
        Space Complexity: O(n) - We store each city
    """

    def greedy(self, time_allowance=60.0):
        bssf = None
        cities = self._scenario.getCities()
        ncities = len(cities)
        startCityDict = {}
        start_time = time.time()

        # Runs until a solution is found or we go over the allotted time
        while (time.time() - start_time) < time_allowance:

            # We will iterate over each city, and explore the best path from starting at that city
            # After the loop we will choose the starting city's route with the lowest cost for our greedy algorithmic solution
            for node in range(len(cities)):

                city = cities[node]
                cityRoute = []
                cityRoute.append(city)
                toVisitCities = deepcopy(cities)
```

```python
                toVisitCities = deepcopy(cities)
                currCity = city

                # We will remove our start city from the array and find the best route from this starting node
                del toVisitCities[node]
                while len(toVisitCities):
                    cityCosts = self.closestCities(currCity, toVisitCities)
                    closestCity = cityCosts[0]
                    locClosestCity = toVisitCities.index(closestCity[0])
                    cityNext = toVisitCities[locClosestCity]


                    cityRoute.append(cityNext)
                    currCity = cityNext

                    # Removes the next city from our array to visit
                    del toVisitCities[locClosestCity]

                if len(toVisitCities):
                    continue
                else:
                    bssf = TSPSolution(cityRoute)
                    end_time = time.time()
                    results = {}
                    results['cost'] = bssf.cost
                    results['time'] = end_time - start_time
                    results['count'] = None
                    results['soln'] = bssf
                    results['max'] = None
                    results['total'] = None
                    results['pruned'] = None
                    startCityDict[node] = results
                    continue

            self.lowestCost = float("inf")
            for key, solution in startCityDict.items():
                if solution["cost"] < self.lowestCost:
                    self.lowestCost = solution["cost"]
                    lowest = solution

        return lowest

    ''' <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>
    '''
```

```python
    """
        Time Complexity: O(n^2 * 2^n) for worst case, but on average closer to O(k * n^2), where k is the number of states checked
        Space Complexity: O(n^2)

        I also implemented the __lt__ operator on the City class that compares indices
            to make comparisons easier. It is constant time.
    """

    def branchAndBound(self, time_allowance=60.0):

        # We are going to use our greedy algorithm to find our initial BSSF
        # This will give us a better starting point than default
        bssf = self.greedy(time_allowance=time_allowance)['soln']

        cities = self._scenario.getCities()
        cities = cities

        ncities = len(cities)
        heap = []
        bssfUpdates = 0
        maxStoredStates = 1
        totalStates = 1
        prunedStates = 0
        numSolutions = 0
        lowestCost = bssf.cost

        # Cost for initializing reduced matrix is O(n^2)
        initReducedMatrix, lowerBound = self.initalReducedMatrix(cities)

        starting = tuple(lowerBound, cities[0], cities[1:],initReducedMatrix,[cities[0]._index],lowerBound)
        heapq.heappush(heap, starting)
        time_start = time.time()

        # We will run until our heap is empty, or we go overe our time-allotment (60 seconds by default)
        while ((time.time() - time_start) < time_allowance and len(heap)):

            # Time complexity for heapq's heappop is O(log(n))
            onDeck = heapq.heappop(heap)

            if onDeck[5] < self.lowestCost:

                # We will iterate through each city we still need to visit
                for city in onDeck[2]:

                    # We need to check if the path event exits (has an edge)
                    if self._scenario._edge_exists[onDeck[1]._index][city._index]:

                        # Creation of our reduced matrix is O(n^2)
                        newSubtree = self.reducedMatrix(city, onDeck[3], onDeck)

                        # checks to see if we have more cities to visit
                        if not len(newSubtree[2]):

```

```python
                            # This will take O(n) in worst case
                            route = self.convertIndicesToCities(newSubtree[4])

                            # checks and creation are O(1) time and space
                            bssf = TSPSolution(route)
                            if bssf.cost < self.lowestCost:
                                self.lowestCost = min(bssf.cost, self.lowestCost)
                                bssfUpdates += 1
                            numSolutions += 1

                        # else if there are no more cities to visit
                        else:

                            # If it isn't lower than lowest cost then we will prune.
                            if newSubtree[5] < self.lowestCost:
                                # Like heappop, heappush is O(log(n)) to reheapify
                                heapq.heappush(heap, newSubtree)
                                totalStates += 1
                            else:
                                prunedStates += 1
                                totalStates += 1


            else:
                prunedStates += 1
                totalStates += 1
            maxStoredStates = max(len(heap), maxStoredStates)


        end_time = time.time()
        results = {}
        results['cost'] = self.lowestCost
        results['time'] = end_time - time_start
        results['count'] = numSolutions
        results['soln'] = bssf
        results['max'] = maxStoredStates

        results['total'] = totalStates
        results['pruned'] = prunedStates

        return results

    def getCost(self, newSubtree):
        value = newSubtree[0]
        citiesRemaining = len(newSubtree[2])
        return value  ## / np.square(self.num_cities - citiesRemaining)

    """
        Time Complexity: O(n)
        Space Complexity: O(n)
    """
```

```python

    def convertIndicesToCities(self, indicesCities):
        cityList = []
        for index in indicesCities:
            cityList.append(self.cities[index])

        return cityList


    """
        Time Complexity: O(n)
        Space Complexity: O(1)
    """

    def clearCity(self, toVisitCities, toVisitNext):
        for index, city in enumerate(toVisitCities):
            if city._index == toVisitNext._index:
                clearIndex = index
                break

        del toVisitCities[clearIndex]
        return toVisitCities

    """
        Time Complexity: O(n)
        Space Complexity: O(n)
    """

    def closestCities(self, city, cityList):
        cost = {}
        for toVisitCities in cityList:
            cost[toVisitCities] = city.costTo(toVisitCities)

        byCostCities = sorted(cost.items(), key=lambda kv: kv[1])
        # print("Closest length is {}".format(byCostCities[0][1]))
        return byCostCities

    """
        Time Complexity: O(n^2) to compare distance of node from every other ndoe
        Space Complexity: O(n^2) to store all relative distances
    """

    def reducedMatrix(self, toVisitNext, matrix, scenario):
        subtree = deepcopy(scenario)
        matrix = matrix.copy()
        reduceSum = 0
        initCost = matrix[subtree[1]._index][toVisitNext._index]


        matrix[subtree[1]._index] = np.inf
        matrix[:, toVisitNext._index] = np.inf
        matrix[toVisitNext._index][subtree[1]._index] = np.inf

```

```python

        # Reduce columns and rows
        for row in range(matrix.shape[0]):
            rowMin = np.min(matrix[row])
            if np.isinf(rowMin):
                continue
            matrix[row] = matrix[row] - rowMin
            reduceSum += rowMin


        for col in range(matrix.shape[1]):
            colMin = np.min(matrix[:, col])
            if np.isinf(colMin):
                continue
            matrix[:, col] = matrix[:, col] - colMin
            reduceSum += colMin

        toVisitCities = subtree[2]
        toVisitCities = self.clearCity(toVisitCities, toVisitNext)

        newCost = subtree[5] + initCost + reduceSum

        # Value for the queue
        newValue = score / len(cities_visited)


        distanceTuple = (newValue, toVisitNext, toVisitCities, matrix, subtree[4] + [toVisitNext._index], newCost)

        return distanceTuple

    """
        Time Complexity: O(n^2) to compare distance of node from every other ndoe
        Space Complexity: O(n^2) to store all relative distances
    """

    def initalReducedMatrix(self, cityList):

        # We will initialize our matrix, with infinity
        matrix = np.full((len(cityList), len(cityList)), fill_value=np.inf)

        # We will find relative distances, all diagnols will remain infinity
        for fromLoc, city in enumerate(cityList):

            for toLoc, toCity in enumerate(cityList):

                if fromLoc == toLoc:
                    # already init to inf
                    continue

                matrix[fromLoc][toLoc] = city.costTo(toCity)

```

```python

        reduceSum = 0
        # do rows and get mins
        for row in range(matrix.shape[0]):
            rowMin = np.min(matrix[row])
            matrix[row] = matrix[row] - rowMin
            reduceSum += rowMin

        # get reduced columns now
        for col in range(matrix.shape[1]):
            colMin = np.min(matrix[:, col])
            matrix[:, col] = matrix[:, col] - colMin
            reduceSum += colMin

        return matrix, reduceSum

    ''' <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>

        Because of complications with Covid-19, we are not going to complete this
        portion of the project
    '''
    def fancy(self, time_allowance=60.0):
        print("Not yet implemented")
```