Lab 5 Report

Tuesday, April 6, 2021 2:04 AM

1. [20] Include your well-commented code.

See the very end of this report for the full code. It is very long so I don't want to clutter the lab report by including it first.

2. [10] Explain both the time and space complexity of your algorithm by showing and summing up the complexity of each subsection of your code. Keep in mind the following things:

- Priority Queue

Here is my priority queue. It is full of StateWrapper objects.

```
188 # Perform branch and bound work on our newly created states
189 #
190 # Time complexity: O(however many states we generate * n^3) because our queue is full of state objects
191 # and for the row in focus O(n) we will perform a statify() O(n^2) operation on every cell. Therefore, we have
192 # a O(n^2) operation inside of an O(n) operation, giving us O(n^3) time for every state in the queue. Therefore,
193 # total time complexity = O(however many states we generate * n^3).
194 #
195 # Space complexity: O(number of states I generate * n^2) Because I have a queue full of every
196 #                   state that I generate. Each state object has a table of size O(n^2) inside of it.
197 while self._queue:
198 
199     # Pop off the state with the smallest bssf/bound thing in the queue
200     state = heapq.heappop(self._queue)
201 
202     # Perform a pruning check
203     # self._results['soln'].cost holds our current BSSF cost
204     if state._state_bound < self._results['soln'].cost:
205 
206         # Get the row to expand into more states
207         row = state._src_path[1]  # 1 Is the col index. Our row we will
208                                   # expand will come from the prev
209                                   # state's column.
210         table = state._table
211 
212         startCity = state._route[0]
213 
214         # Expand the cell in the row into a new state if it is not infinity
215         for i in range(len(table[row])):
216             if table[row][i][0] != INFINITY:  # 0 because that is where the cost to get to
217                                               # the city is stored
218                 # Make sure we don't re-visit cities, but we are ok to revisit the start city
219                 # if we are in the last city
220                 if table[row][i][1] != startCity or len(state._route) == len(self._cities):
221                     self.statify(row, i, table, state._route, state._depth, state._state_bound)
222     else:  # We need to prune this state
223         self._results['pruned'] += 1
224 
225     # Do we have time to do more work?
226     # This check happens after we build states from every row in focus
227     self.timeCheck()
228     if self._timesUp:
229         return self.wrapThingsUp()
```

Time Complexity: See the comments above the while loop. Processing the queue is the most time complex part of my branch and bound algorithm because it causes me to perform (worst case) O(n^2) time complexity operation from my statify() function inside of a O(n) for loop, resulting in O(n^3) time. We perform this O(n^3) operation for every state in our queue, therefore, the total time complexity is O(however many states we generate * n^3).

Space Complexity: The queue holds state objects that have O(n^2) space complexity because they hold a table of n rows and no columns. This combined with the number of states inside of the queue makes the total space complexity to be O(number of states * n^2).

- SearchStates

The above code snipped shows how I search through my states. On line 200, I pop the state with the lowest bound and the deepest depth off of the queue. If the bound is less than the current BSSF, I deem it worthy of expanding its children. I then expand every child state from the row in focus using the code in lines 215-222.

Time Complexity: O(n^3) because statify() is O(n^2) time complexity and I call statify() inside of a O(n) for loop.

Space Complexity: The space complexity of code lines 215-222 is O(number of generated states * n^2). This is because statify() generates a state object that is of size O(n^2) (thanks to a table of n rows and n columns), and adds it to the queue.

- Reduced Cost Matrix, and updating it

```python
# Performs operations on the parent table to update the bounds
# from moving to a new city, and creates a table to show those
# changes.
# Time Complexity: O(n^2) because our zeroRows(), zeroCols(),
#                  and infinitize() functions are all O(n^2).
# Space Complexity: O(n^2) because the table object is the
#                   largest object at O(n^2) (n rows and n cols).
def statify(self, row, col, table, route, depth, bound):

    # Create a table that matches the parent table
    tableMatch = []
    for i in range(len(table)):
        tableMatch.append([])
        for j in range(len(table)):
            tableMatch[i].append(table[i][j])

    # Inifinitize the rows and columns in the table from the operation
    updatedBound, tableMatch, solnFound = self.infinitize(row, col, tableMatch, bound)

    # Zero out the rows. Update the bound & table
    updatedBound, tableMatch = self.zeroRows(updatedBound, tableMatch)

    # Zero out the cols. Update the bound & table
    updatedBound, tableMatch = self.zeroCols(updatedBound, tableMatch)
```

The above code snipped shows where I build the reduced cost matrix for every child state. I build the reduced cost matrix from the child's parent's reduced cost matrix. The total time and space complexity for updating the reduced cost matrix is O(n^2). The infinitize(), zeroRows(), and zeroCols() functions make up this time and space complexity. An analysis of each function's time and space complexity follows below until the end of this sections bullet point.

```python
# Updates the rows and cols to be infinity based on the city path taken to get there.
# Updates the backtrace to be infinity
#
# Time Complexity: O(n^2). We have a double for loop to check every cell in the table.
# Space Complexity: O(n^2). The table is of size O(n^2)
def infinitize(self, row, col, table):

    infinityCount = 0
    doneCount = len(table) * len(table)

    # O(n^2) loops
    for i in range(len(table)):
        for j in range(len(table[i])):
            if table[i][j][0] == INFINITY:  # 0 because the table is of tuples (cost, city)
                infinityCount += 1
            elif row == i:  # Make the row infinity (if qualifies)
                table[i][j] = (INFINITY, table[i][j][1])
            elif col == j:  # Make the col infinity (if qualifies)
                table[i][j] = (INFINITY, table[i][j][1])
```

The above code snippet shows the infinitize() function. This is not all of the code in the function, but it is the code that makes the time and space complexity what it is. This affects the updating of the reduced cost matrix time comlpexity because the infinitize() function is called when updating the reduced cost matrix. This infinitize() function makes the reduced cost matrix update have a time complexity of O(n^2) because of its double for loops and a space complexity of O(n^2) because of its table of n rows and n columns.

```python
# Subtracts the min value in every row from every element
# to ensure we always have at least one zero in every row.
# Adds the subtraction amount to the bound and returns it.
#
# Time complexity: O(n^2) because we look at every cell in table.
# Space Complexity: O(n^2) because we have a table of n rows and n columns.
def zeroRows(self, bound, table):

    for i in range(len(table)):
        minVal = INFINITY

        # Find row values
        for j in range(len(table[i])):
            if table[i][j][0] < minVal:  # 0 because the table is of tuples (cost, city)
                minVal = table[i][j][0]

        # If a minVal < INFINITY exists then we should update our table
        # Update the bound
        if minVal != INFINITY:
            bound += minVal

            # Update row values
            for j in range(len(table[i])):
                tempVal = table[i][j][0]
                tempVal -= minVal
                table[i][j] = (tempVal, table[i][j][1])  # 0 because the table is of tuples (cost, city)

    return bound, table
```

Above is the zeroRows() function. This function, like the infinitize() function affects the reduced cost matrix update because of its O(n^2) time complexity (thanks to the double for loop), and because of its O(n^2) space complexity (thanks to its table of n rows and n columns.

- BSSF Initialization

```python
# Time complexity: O(n^2) because we compare every city to every other city when
#                  we are trying to find the min cost.
# Space complexity: O(n). Because of a map of size n called visitedCities,
#                   a heapQueue that is of size n (worst case),
#                   and a route list of size n.
def greedy(self, time_allowance=60.0):
    results = {}
    cities = self._scenario.getCities()
    visitedCities = {}

    startTime = time.time()
    route = []
    route.append(cities[0])  # Add starting city to the route
    visitedCities[cities[0]._name] = True  # Prevent us from visiting our starting city

    runningCost = 0

    # For every city, get the minimal cost to another city and add it to our route
    i = 0
    counter = 0
    # This is O(n^2) because of an n size loop inside of an n size loop
    while (counter < len(cities)):
        currCity = cities[i]
        heapFromCurrCity = []

        # Find costs from current city to every other city
        for j in range(len(cities)):
            tempCity = cities[j]
            try:
                isVisited = visitedCities[tempCity._name]  # Will not throw exception if city has
                # been visited. Therefore, skip it.
            except:
                cost = currCity.costTo(tempCity)
                wrappedCity = CityWrapper(cost, tempCity, j)
                heapq.heappush(heapFromCurrCity, wrappedCity)

        # Obtain the closest city, make it impossible to visit in the future, and add it to the route.
        if len(heapFromCurrCity) == 0:
            # currCity is the last city. We are done.
            break
        wrappedCity = heapq.heappop(heapFromCurrCity)
        closestCityToCurrentCity = wrappedCity._city
        cost = wrappedCity._cost
        i = wrappedCity._indexInCities
        visitedCities[closestCityToCurrentCity._name] = True  # Prevent us from visiting the closest city
        # in future calculations
        route.append(closestCityToCurrentCity)
        runningCost += cost

        counter += 1
```

My branch and bound BSSF is constructed from this greedy() function. This is not the entire code for my function, but it is all of the code that affects the time and space complexity of finding the BSSF.

Time Complexity: O(n^2) because I compare every city to every other city.
Space Complexity: O(n) because I build a map (visitedCities) of size n, a queue (named heapFromCurrCity) of size n, and a route (named route) of size n.

- Expanding one SearchState into others

See the code snippet used in my explanation of "Priority Queue" in the above bullet points. I expand the SearchState into others in lines 215-222.

Time Complexity: O(n^3) because I perform statify() O(n^2) inside of a O(n) for loop.

Space Complexity: O(n^2) because I work with tables of size O(n^2) (n rows and n columns)

- The full Branch and Bound algorithm

Given the time and space complexities of each subsection of code utilized in my full Branch and Bound Algorithm, the algorithm has the following time and space complexities:

Time Complexity: O(number of states I generate * n^3) Because I have a O(n^3) time complex operation to generate and build each state, and then I process every state in my queue.

Subsections that affect this time complexity: Priority Queue and Expanding one SearchState into others,

Space Complexity: O(number of states I generate * n^2) Because I have a queue full of every state that I generate. Each state object has a table of size O(n^2) inside of it.

Subsections that affect this space complexity: Priority Queue, BSSF Initialization, updating the Reduced Cost Matrix

3. [5] Describe the data structures you use to represent the states.

- StateWrapper.py
  - This is a class I created to represent state objects
  - This class stores the following:
    - A reduced cost matrix for the state
    - The city path taken to expand into this state
    - The full city route taken to arrive at this state
    - The bound (or cost) of making it to this state
    - The dept of the state in our tree
  - This class also overwrites the less than (__lt__ (self, other)) function which helps the priority queue know that it should always store the state with the smallest bound and deepest depth at the top of the heap.

4. [5] Describe the priority queue data structure you use and how it works.

- Priority Queue
  - The priority queue holds a min heap of StateWrapper objects.
  - The priority queue is implemented using the heapq library.
  - The state to be popped off is always the state with the lowest bound cost and deepest tree depth.

5. [5] Describe your approach for the initial BSSF.

I used the greedy algorithm for my BSSF.

My greedy algorithm compares each city to every other city. It takes the smallest path, makes the current city unvisitable, and then tries to find the next shortest path for the new current city.

My greedy algorithm function uses a heapq of CityWrapper objects. It takes O(n^2) time to find the correct path, but takes only O(1) time to retrieve the correct path after having found it thanks to the heapq pop() method.

The CityWrapper object is a simple object that contains the cost from the current city used to get to that city, the name of the city, and the city's index in the overall cities list.

6. [25] Include a table containing the following columns.

# 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	8.457	3481	310	11	7481	6313
16	902	14.228	3561	294	6	11434	9874
50	33	60.7517	17510	1033	1	1087	1033
50	17	60.1615	12647	995	3	1068	1009
45	3	60.8465	15258	1420	1	1494	1421
45	5	60.4871	17336	1302	1	1375	1303
10	5	4.161	4264	66	4	1278	912
10	33	0.5	2036	30	1	209	162
20	33	60.0103	5329	939	3	5736	5048
6	666	0.06	1898	11	2	28	14
7	777	0.15	2402	14	3	82	47
14	777	27.6711	3817	158	5	5211	4323
14	55	4.6467	2567	95	4	833	686
28	55	60.242	7997	2070	4	2740	2458

7. [10] Discuss the results in the table and why you think the numbers are what they are, including how time complexity and pruned states vary with problem size.

My results in the table have very few numbers of BSSF updates for times that are >= 60 seconds. I believe that this is the case because I prioritize depth in my queue (along with minimal bounds). My BSSF only has a chance to be updated once a tree edge's reduced cost matrix is full of infinities. Due to this, I believe my long executing runtimes have fewer BSSF updates because it takes them longer to come to a potential solution, therefore, there is less opportunity to discover other potential BSSFs.

My time complexity varies with different problem sizes. Larger problem sizes require more states to be generated, expanded, and pruned. This takes more time. Therefore, the larger the problem size the longer the runtime is (obviously this is seed dependent as some seed solutions are closer to their BSSF)

My pruned states also correlate with problem size. The larger the problem is, the more states I will generate, therefore I have more opportunities to prune states. It is true that my BSSF updates are typically lower with the larger solutions, but this does not mean that my initial BSSF would not be pruning states.

8. [10] Discuss the mechanisms you tried and how effective they were in getting the state space search to dig deeper and find more solutions early.

I made my state space search dig deeper by prioritizing tree depth together with minimum bounds when I would select a state to expand from my queue. By doing this, I guaranteed that I would be expanding the cheapest solutions first and seeing them through to the end to see if they would update the BSSF. Doing this ensured that I would finding the optimal solution in the quickest time possible by pruning as many states as possible at the beginning (thus decreasing future potential for state expansion).

My Code:

TSPSolver

File - C:\Users\jacob\Documents\Winter2021\312\Labs\LabFive\TSPSolver.py

```
#!/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 *
from CityWrapper import *
from StateWrapper import *
import heapq
import itertools
import copy

INFINITY = math.inf


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 = {}
        self._cities = self._scenario.getCities()
        ncities = len(self._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(self._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 INFINITY
        results['time'] = end_time - start_time
        results['count'] = count  # Number of solutions
discovered.
        results['soln'] = bssf  # Object containing the route.
        results['max'] = None  # Max size of the queue.
        results['total'] = None  # Total states generated.
        results['pruned'] = None  # Number of states pruned.
        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^2) because we compare every city to
every other city when
    #                  we are trying to find the min cost.
    # Space complexity: O(n). Because of a map of size n called
 visitedCities,
    #                   a heapQueue that is of size n (worst
case),
    #                   and a route list of size n.
    def greedy(self, time_allowance=60.0):

        results = {}
        self._cities = self._scenario.getCities()
        visitedCities = {}

        self._startTime = time.time()
        self._time_allowance = time_allowance
        route = []
        route.append(self._cities[0])  # Add starting city to
the route
        visitedCities[self._cities[0]._name] = True  # Prevent
us from visiting our starting city

        runningCost = 0

        # For every city, get the minimal cost to another city
and add it to our route
        i = 0
        counter = 0
        # This is O(n^2) because of an n size loop inside of an
 n size loop
        while (counter < len(self._cities)):
            currCity = self._cities[i]
            heapFromCurrCity = []

            # Find costs from current city to every other city
            for j in range(len(self._cities)):
```

```
                tempCity = self._cities[j]
                try:
                    isVisited = visitedCities[tempCity._name]
# Will not throw exception if city has
                    # been visited. Therefore, skip it.
                except:
                    cost = currCity.costTo(tempCity)
                    wrappedCity = CityWrapper(cost, tempCity, j
)
                    heapq.heappush(heapFromCurrCity,
wrappedCity)

            # Obtain the closest city, make it impossible to
visit in the future, and add it to the route.
            if len(heapFromCurrCity) == 0:
                # currCity is the last city. We are done.
                break
            wrappedCity = heapq.heappop(heapFromCurrCity)
            closestCityToCurrentCity = wrappedCity._city
            cost = wrappedCity._cost

            i = wrappedCity._indexInCities
            visitedCities[closestCityToCurrentCity._name] =
True  # Prevent us from visiting the closest city
            # in future calculations
            route.append(closestCityToCurrentCity)
            runningCost += cost
            counter += 1

        endTime = time.time()
        bssf = TSPSolution(route)
        bssf.cost = INFINITY
        if len(visitedCities.keys()) == len(self._cities):
            bssf.cost = runningCost

        timePassed = endTime - self._startTime

        results['time'] = timePassed
        results['cost'] = bssf.cost
        results['count'] = 0  # Number of solutions discovered
. Will always be 1 for the greedy solution
        results['soln'] = bssf  # Object containing the route.
        results['max'] = 0  # Max size of the queue. Will
always be 1 for the greedy solution
```

```
        results['total'] = 0  # Total states generated. Will
always be 1 for the greedy solution
        results['pruned'] = 0  # Number of states pruned. Will
always be 0 for the greedy solution

        return results

    ''' <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>
    '''

    # Time Complexity: O(number of states I generate * n^3)
Because I have a O(n^3) time
    #                  complex operation to generate and build
each state, and then I
    #                  process every state in my queue.
    # Space Complexity: O(number of states I generate * n^2)
Because I have a queue full of every
    #                   state that I generate. Each state
object has a table of size O(n^2) inside
    #                   of it.
    def branchAndBound(self, time_allowance=60.0):
        # Start the timer
        self._startTime = time.time()
        self._timesUp = False

        self.initResults()
        bound, table = self.createParent()
        self._queue = []

        # Add the first state to the queue
        #             StateWrapper(matrix, bound, srcPath,
route, depth)
        firstCity = self._cities[0]
```

```
        returnState = StateWrapper(table, bound, (0, 0), [
firstCity], 0)
        heapq.heappush(self._queue, returnState)
        self._results['max'] += 1 # Because we now have our
first state in the queue
        self._results['total'] += 1  # Because we have
generated a state


        # Perform branch and bound work on our newly created
states
        #
        # Time complexity: O(however many states we generate
 * n^3) because our queue is full of state objects
        # and for the row in focus O(n) we will perform a
statify() O(n^2) operation on every cell. Therefore, we have
        # a O(n^2) operation inside of an O(n) operation,
giving us O(n^3) time for every state in the queue. Therefore,
        # total time complexity = O(however many states we
generate * n^3).
        #
        # Space complexity: O(number of states I generate * n^
2) Because I have a queue full of every
        #                           state that I generate. Each state
object has a table of size O(n^2) inside of it.
        while self._queue:

            # Pop off the state with the smallest bssf/bound
thing in the queue
            state = heapq.heappop(self._queue)

            # Perform a pruning check
            # self._results['soln'].cost holds our current BSSF
 cost
            if state._state_bound < self._results['soln'].cost:

                # Get the row to expand into more states
                row = state._src_path[1]  # 1 Is the col index
. Our row we will
                                          # expand will come
from the prev
                                          # state's column.
                table = state._table
```

```
                startCity = state._route[0]

                # Expand the cell in the row into a new state
 if it is not infinity
                for i in range(len(table[row])):
                    if table[row][i][0] != INFINITY:  # 0
 because that is where the cost to get to
                                                      # the
 city is stored
                        # Make sure we don't re-visit cities,
 but we are ok to revisit the start city
                        # if we are in the last city
                        if table[row][i][1] != startCity or len
(state._route) == len(self._cities):
                            self.statify(row, i, table, state.
_route, state._depth, state._state_bound)
            else:  # We need to prune this state
                self._results['pruned'] += 1

            # Do we have time to do more work?
            # This check happens after we build states from
 every row in focus
            self.timeCheck()
            if self._timesUp:
                return self.wrapThingsUp()

        # Done processing all of our states
        return self.wrapThingsUp()

    def wrapThingsUp(self):
        # Set the time it took to perform the algorithm
        currTime = time.time()
        tempTimePassed = currTime - self._startTime
        self._results['time'] += tempTimePassed
        self._results['cost'] = self._results['soln'].cost
        print("Time: " + str(self._results['time']))
        self._results['pruned'] += len(self._queue)  # Add
 states that were never processed to the pruned count.
                                                     # The lab
  specs ask us to do this.

        # Done
        return self._results
```

```
    def timeCheck(self):
        currTime = time.time()
        tempTimePassed = currTime - self._startTime
        totalTimePassed = tempTimePassed + self._results['time
']
        if totalTimePassed >= self._time_allowance:
            self._timesUp = True

    # Performs operations on the parent table to update the
bounds
    # from moving to a new city, and creates a table to show
those
    # changes.
    # Time Complexity: O(n^2) because our zeroRows(), zeroCols
(),
    #                  and infinitize() functions are all O(n^2
).
    # Space Complexity: O(n^2) because the table object is the
    #                   largest object at O(n^2) (n rows and n
cols).
    def statify(self, row, col, table, route, depth, bound):

        # Create a table that matches the parent table
        tableMatch = []
        for i in range(len(table)):
            tableMatch.append([])
            for j in range(len(table)):
                tableMatch[i].append(table[i][j])

        # Inifinitize the rows and columns in the table from
the operation
        updatedBound, tableMatch, solnFound = self.infinitize(
row, col, tableMatch, bound)

        # Zero out the rows. Update the bound & table
        updatedBound, tableMatch = self.zeroRows(updatedBound,
tableMatch)

        # Zero out the cols. Update the bound & table
        updatedBound, tableMatch = self.zeroCols(updatedBound,
tableMatch)

        # Add the path taken to get to the city to the route
        updatedRoute = copy.deepcopy(route)
```

```
        updatedRoute.append(tableMatch[row][col][1])  # 1
because that is the city index in the tuple in the table

        # Update things if we have found a solution and it is a
 better solution
        if solnFound and updatedBound < self._results['soln'].
cost:
            # Update the count of solutions discovered.
            self._results['count'] += 1
            # Set the BSSF to help with future pruning.
            self._results['soln'] = TSPSolution(route) # Use
the original route so it doesn't have the first city twice.
            self._results['soln'].cost = updatedBound
            self._results['pruned'] -= 1 # Just because it is
going to increment one above what
                                          # what it should be
right after this on line 303.
            self._results['total'] -= 1 # Similar situation to
the pruned problem above...

        # The updatedBound is < BSSF so it is worth pursuing
this route
        if updatedBound < self._results['soln'].cost:
            # Create variables to create a state to add to the
queue
            srcPath = (row, col)
            returnState = StateWrapper(tableMatch, updatedBound
, srcPath, updatedRoute, depth + 1)

            # Add the state object to the queue and see if our
queue
            # is the biggest it has ever been
            heapq.heappush(self._queue, returnState)
            if self._results['max'] < len(self._queue):
                self._results['max'] = len(self._queue)
        else:
            self._results['pruned'] += 1

        # Update the our counter for tracking the number
        # of generated states
        self._results['total'] += 1

    # Subtracts the min value in every row from every element
    # to ensure we always have at least one zero in every row.
```

```
    # Adds the subtraction amount to the bound and returns it.
    #
    # Time complexity: O(n^2) because we look at every cell in
table.
    # Space Complexity: O(n^2) because we have a table of n
rows and n columns.
    def zeroRows(self, bound, table):

        for i in range(len(table)):
            minVal = INFINITY

            # Find row values
            for j in range(len(table[i])):
                if table[i][j][0] < minVal:  # 0 because the
table is of tuples (cost, city)
                    minVal = table[i][j][0]

            # If a minVal < INFINITY exists then we should
update our table
            # Update the bound
            if minVal != INFINITY:
                bound += minVal

                # Update row values
                for j in range(len(table[i])):
                    tempVal = table[i][j][0]
                    tempVal -= minVal
                    table[i][j] = (tempVal, table[i][j][1])
# 0 because the table is of tuples (cost, city)

        return bound, table

    # Subtracts the min value in every col from every element
    # to ensure we always have at least one zero in every col.
    # Adds the subtraction amount to the bound and returns it.
    #
    # Time complexity: O(n^2) because we look at every cell in
table.
    # Space Complexity: O(n^2) because we have a table of n
rows and n columns.
    def zeroCols(self, bound, table):

        for i in range(len(table)):
            minVal = INFINITY
```

```
            # Find col values
            for j in range(len(table)):
                if table[j][i][0] < minVal:  # 0 because the
 table is of tuples (cost, city)
                    minVal = table[j][i][0]

            # If a minVal < INFINITY exists then we should
 update our table
            # Update the bound
            if minVal != INFINITY:
                bound += minVal

                # Update col values
                for j in range(len(table[i])):
                    tempVal = table[j][i][0]
                    tempVal -= minVal
                    table[j][i] = (tempVal, table[j][i][1])
 # 0 because the table is of tuples (cost, city)

        return bound, table

    # Updates the rows and cols to be infinity based on the
 city path taken to get there.
    # Updates the backtrace to be infinity
    #
    # Time Complexity: O(n^2). We have a double for loop to
 check every cell in the table.
    # Space Complexity: O(n^2). The table is of size O(n^2)
    def infinitize(self, row, col, table, bound):

        infinityCount = 0
        doneCount = len(table) * len(table)

        # Add the current cell's value to the bound
        bound += table[row][col][0]

        # O(n^2) loops
        for i in range(len(table)):
            for j in range(len(table[i])):
                if table[i][j][0] == INFINITY:  # 0 because the
 table is of tuples (cost, city)
                    infinityCount += 1
                elif row == i:  # Make the row infinity (if
```

```
qualifies)
                    table[i][j] = (INFINITY, table[i][j][1])
                    infinityCount += 1
                elif col == j:  # Make the col infinity (if
qualifies)
                    table[i][j] = (INFINITY, table[i][j][1])
                    infinityCount += 1

        # Infinitize the backtrace
        if table[col][row][0] != INFINITY:
            table[col][row] = (INFINITY, table[col][row][1])
# Handles the reverse of table[row][col]
            infinityCount += 1

        solnFound = False
        if infinityCount == doneCount:
            solnFound = True

        return bound, table, solnFound

    # Time Complexity: O(n^2) because we compare every city to
every other city.
    # Space Complexity: O(n^2) because we create a table of n
rows and n columns.
    def createParent(self):
        self._cities = self._scenario.getCities()
        numCities = len(self._cities)
        table = []

        # Create empty table to fill
        for city in self._cities:
            table.append([])

        # Populate the parent table
        for i in range(numCities):
            currCity = self._cities[i]
            for j in range(numCities):
                cost = currCity.costTo(self._cities[j])
                table[i].append((cost, self._cities[j]))

        # Find the bound and create the reduced cost matrix
        # This takes O(n^2) time complexity and O(n^2) space
complexity
        bound, table = self.zeroRows(0, table)
```

File - C:\Users\jacob\Documents\Winter2021\312\Labs\LabFive\TSPSolver.py

```
        bound, table = self.zeroCols(0, table)

        return bound, table

    # Time Complexity: O(n^2) because my greedy solution is of
 O(n^2) complexity
    # Space Complexity: O(n^2) because my greedy solution
 creates a list of cities
    #                   in its route.
    def initResults(self):
        self._results = self.greedy()

    ''' <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
```

Page 13 of 13

CityWrapper

File - C:\Users\jacob\Documents\Winter2021\312\Labs\LabFive\CityWrapper.py

```
from TSPClasses import *

class CityWrapper:

    def __init__(self, cost, city, indexInCities):
        self._cost = cost
        self._city = city
        self._indexInCities = indexInCities

    def __lt__(self, other):
        return self._cost < other._cost
```

Page 1 of 1

StateWrapper

File - C:\Users\jacob\Documents\Winter2021\312\Labs\LabFive\StateWrapper.py

```
from TSPClasses import *

class StateWrapper:

    def __init__(self, table, state_bound, src_path, route,
depth):
        self._table = table
        self._state_bound = state_bound
        self._src_path = src_path
        self._route = route
        self._depth = depth

    def __lt__(self, other):
        if self._depth > other._depth:
            return True
        elif self._state_bound < other._state_bound:
            return True
        else:
            return False
```

Page 1 of 1

TSPClasses

Note: I included the TSPClasses.py file because I made some changes to the City class. Changes included overwriting __str__ and __eq__ functions

File - C:\Users\jacob\Documents\Winter2021\312\Labs\LabFive\TSPClasses.py

```
#!/usr/bin/python3


import math
import numpy as np
import random
import time



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
        last = self.route[0]
        for city in self.route[1:]:
            cost += last.costTo(city)
            last = city
```

```
        cost = 0
        last = self.route[0]
        for city in self.route[1:]:
            cost += last.costTo(city)
            last = city
        cost += self.route[-1].costTo( self.route[0] )
        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 ) )
            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

        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)

        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



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

    def __str__(self):
        return self._name

    def __eq__(self, other):
        if self._name == other._name:
            return True
        return False

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

    def setScenario( self, scenario ):
        self._scenario = scenario

    ''' <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]:
            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                  # Shouldn't it cost
 something to go downhill, no matter how steep??????


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