Project 5: Traveling Salesperson

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

```
def branchAndBound(self, time_allowance=60.0):
    results = {}
    heap = []
    hq.heapify(heap)

    cities = self._scenario.getCities()
    self.printCities(cities, "cities:")
    costMatrix = self.createMatrix()  # creates 2d matrix of nxn where n is the length of cities array

    result = self.reduceMatrix(0,costMatrix)  # caclculates the lower cost bound and does reeduction matrix operation (rows than columns)
    # O (n^2)


    # Initial state
    s = State(cities[0]._name)
    s.matrix = result[0]
    s.lower_bound = result[1]
    s.markCity(cities[0]._index)
    s.index = cities[0]._index

    # Key of priority queue is an integer. We conver float using int().
    # We divide lower bound by the number of cities this will force the algorithm go down instead of side.
    # Key is lower cost bound divided by the depth.
    hq.heappush(heap, (int(result[1] / len(s.getVisited())), s))

    # We use a random algorithm to find a random initial BSSF that we will initially use.
    self.bssf = self.defaultRandomTour()["cost"]

    # We convert the array of Cities objects into an array of indices since the number of cities won't change
    listInd = self.getCityIndexList(cities)
    cost = None
    tour = []
     foundTour = False

     # s*L = n! n^3 + log(max size of queue) = max size of queue = n! =>  O(n!n^3)
     # space is max size of queue (n!) n!n^2(space of matrix)
     start_time = time.time()
     while len(heap) != 0 and time.time() - start_time < time_allowance:
         wholeState = hq.heappop(heap) #We convert the first city in queue which is State 1
         state = wholeState[1]
         cost = state.lower_bound
         if cost > self.bssf:
             self.statesPruned += 1
             continue
         else:
             notVisited = self.getCitiesNotVisited(state.getVisited(), listInd)
             # We get the number of cities that the state has not visited O(n)

             if (len(notVisited) == 0):
                 if (self.bssf > cost):
                     lastCity = self.getCitites(state.getVisited())[-1]
                     firstCity = self.getCitites(state.getVisited())[0]
                     path = lastCity.costTo(firstCity)
                     if (path != float("inf")): # it is a solution if there is a path between last and first cities
                         foundTour = True
                         self.BSSFUpdates += 1  # Number of BSSF updates
                         self.bssf = cost # We update BSSF
                         tour = self.getCitites(state.getVisited())
             else:
                 self.modifyHeapQueue(wholeState, notVisited, heap)

     self.printCities(tour, "tour:")
     bssf = TSPSolution(tour) # creation of object containing our solution
     end_time = time.time()  # optimal
```

```
        if (len(tour) != 0):
            results['cost'] = bssf.cost if foundTour else self.bssf  # bssf.cost ???
            results['time'] = end_time - start_time
            results['count'] = self.BSSFUpdates  # number of intermediate solutions considered ???
            results['soln'] = bssf
            results['max'] = self.storedStates
            results['total'] = self.statesCreated
            results['pruned'] = self.statesPruned  # Total # of states pruned
            return results

        else:
            results['cost'] = self.bssf
            results['time'] = end_time - start_time
            results['count'] = self.BSSFUpdates  # number of intermediate solutions considered ???
            results['soln'] = bssf  # should we return the alretative ???
            results['max'] = self.storedStates
            results['total'] = self.statesCreated
            results['pruned'] = self.statesPruned  # Total # of states pruned
            return results

    def createMatrix(self):
        cities = self._scenario.getCities()
        rows = len(cities)
        cols = len(cities)

        costMatrix = [[]]
        costMatrix = [[0 for i in range(rows)] for j in range(cols)]

        for i in range(len(costMatrix)):
            city = cities[i]  # city A
            for j in range(len(costMatrix[i])):  # cities A B C D E F
                costMatrix[i][j] = city.costTo(
                    cities[j])  # cost from(A) --> to(A), cost from(A) --> to(B), A --> C, A --> D, ...
        # self.printConstMatrix(costMatrix)

        return costMatrix

    def reduceMatrix(self, lowerBound, matrix):
        cities = self._scenario.getCities()
        bound_cost = lowerBound
        for i in range(len(matrix)):
            min_val = min(matrix[i])

            if (min_val == 0 or min_val == float("inf")):
                continue
            else:
                bound_cost += min_val
                for j in range(len(cities)):
                    cell_val = matrix[i][j]
                    if (cell_val != float("inf")):
                        matrix[i][j] = cell_val - min_val
        for j in range(len(cities)):
            min_val = min([row[j] for row in matrix])
            if (min_val == 0 or min_val == float("inf")):
                continue
            else:
                bound_cost += min_val
                for i in range(len(matrix)):
                    cell_val = matrix[i][j]
                    if (cell_val != float("inf")):
                        matrix[i][j] = cell_val - min_val
        # self.printMatrix(matrix)

        return matrix, bound_cost

    def modifyHeapQueue(self, wholeState, needToVisit, heap):
        cities = self._scenario.getCities()
        parentState = wholeState[1]
        for index in needToVisit:
            originalCost = copy.deepcopy(parentState.lower_bound)
            childMatrix = copy.deepcopy(parentState.getMatrix())
            if (childMatrix[parentState.index][index] == float("inf")):
                self.statesPruned += 1  # Pruned states not added to the queue or not counted becase state not creaated
                continue
            else:
                originalCost = originalCost + childMatrix[parentState.index][index]  # cost of path
                childMatrix = self.markRowsAndColsAndRefl(childMatrix, parentState.index, index)
                result = self.reduceMatrix(originalCost, childMatrix)  # created
                self.statesCreated += 1

                if (result[1] > self.bssf):
                    self.statesPruned += 1
```

```
297
298     def modifyHeapQueue(self, wholeState, needToVisit, heap):
299         cities = self._scenario.getCities()
300         parentState = wholeState[1]
301         for index in needToVisit:
302             originalCost = copy.deepcopy(parentState.lower_bound)
303             childMatrix = copy.deepcopy(parentState.getMatrix())
304             if (childMatrix[parentState.index][index] == float("inf")):
305                 self.statesPruned += 1  # Pruned states not added to the queue or not counted becase state not creaated
306                 continue
307             else:
308                 originalCost = originalCost + childMatrix[parentState.index][index]  # cost of path
309                 childMatrix = self.markRowsAndColsAndRefl(childMatrix, parentState.index, index)
310                 result = self.reduceMatrix(originalCost, childMatrix)  # created
311                 self.statesCreated += 1
312
313                 if (result[1] > self.bssf):
314                     self.statesPruned += 1
315                     continue
316                 else:
317                     s = State(cities[parentState.index]._name)
318                     s.matrix = result[0]
319                     s.lower_bound = result[1]
320                     s.buildName(cities[index]._name)
321                     s.visited = copy.deepcopy(parentState.visited)
322                     s.markCity(cities[index]._index)
323                     s.index = index
324                     # s.show()
325                     hq.heappush(heap, (
326                     int(result[1] / len(s.getVisited())), s))  # we add the state with the lower bound cost to the heap
327                     if (len(heap) > self.storedStates):
328                         self.storedStates = len(heap)
329
330     def enumerate(sequence, start=0):
331         n = start
332         for elem in sequence:
333             yield n, elem
334             n += 1
335
336     def getCitites(self, indices):
337         tour = []
338         cities = self._scenario.getCities()
339         for i in indices:
340             tour.append(cities[i])
341         return tour
342
343     def getCityIndexList(self, cities):
344         indecies = []
345         for city in cities:
346             indecies.append(city._index)
347
348         return indecies
349
350     def getCitiesNotVisited(self, visited, cities):
351         set_difference = set(visited).symmetric_difference(set(cities))
352         list_difference = list(set_difference)
353         return list_difference
354
355     def markRowsAndColsAndRefl(self, matrix, row, col):
356         matrix[col][row] = float("inf")
357         # matrix[:,col] = float("inf") # matrix[:,col] means select rows from column col
358         for i in range(len(matrix)):
359             matrix[i][col] = float("inf")
360
361         for j in range(len(matrix)):
362             matrix[row][j] = float("inf")
363
364         # self.printTheMatrix(matrix)
365         return matrix
```

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.**
In this complexity analysis n is the total number of cities in our problem. The complexity for the full branch and bound is $O(n!n^3)$. That complexity comes from the different elements used in the algorithm. Depending on how we find an initial BSSF the cost could vary a little. If we use an assign BSSF a random value like ∞ could be constant time or linear if we use a random algorithm to find an initial value for BSSF. As it shows in the code above, the algorithm iterates until the priority queue is empty or we run out of time (60 seconds). In the very worst-case scenario, our queue could have all n factorial number of cities which would be O(n!). This priority queue we use for the algorithm organizes the states based on their key which is the lower bound of the state divided by number of cities visited in the state. Getting the smallest value is constant time, O(1), and the cost reorganizing the queue, bubbling up/down, would be the log(max size of the queue) where max size of queue is n! as well. Thus, the queue has a cost of log(n!). Another big cost we have in the algorithm is calculating and creating the new states (searching the states) that we need to visit or need to be added to the queue. Once we have a parent state that we need to create its children we get the difference between the cities visited in the parent state and the total number of states. For each of these cities we need to calculate the lower bound cost and we need to reduce the matrix which is $O(n^2)$. Thus we can see that we have a cost of $O(n! * (n^2) + \log(n!))$ or $O(n!n^2 + n!\log(n!))$ = **$O(n!n^2)$.** The space used for this algorithm is the cost we have for number of times we go through the algorithm, and we create the objects used in the algorithm. The biggest space cost comes from the matrix. Thus, the space complexity is n! and n^2, **$O(n!n^2)$**.
3. [5] **Describe the data structures you use to represent the states.**
The structure that the algorithm uses for the states is a class that stores different information necessary to calculate and represent the "tree" of cities. This class stores an integer that represents a city from the list of cities. This state class has a reduced matrix that the different children of that state will use as the starting matrix. This class also contains the lower bound cost which is the cost of the parent of that state plus the cost of the reduction of the state's matrix. State class also stores an array of cities that have been visited so far. The children inherent this array and add themselves to the array. All that information is used to find a BSSF that can be better than initial BSSF.
4. [5] **Describe the priority queue data structure you use and how it works.** You can use your previously implemented PQ code, or any version available, but you need to describe it.
The priority queue used for the algorithm is the one provided by Python library. This is a binary tree for which every parent node has a value less than or equal to any of its children. The documentation of this heap queue states that "the implementation uses arrays for which heap[k] <= heap[2*k+1} and heap[k] = heap[2*k+2] for all k, counting elements from zero." This implementation assigns infinite for non-existing elements, this is done so the comparison between elements can be done.
5. [5] **Describe your approach for the initial BSSF.**
At first, I assigned a random value to the BSSF variable. This value was not very efficient because I didn't know how big this random value had to be and I was making this BSSSF very low. This cause problems to the algorithm because all the were getting pruned because the value was not possible and was very low even the best cost calculated by the algorithm could not beat this low random value. Then, I decided to use infinite as my initial BSSF value. This was also inefficient because since the algorithm never has a lower bound cost value of infinite which

makes the BSSF to get updated in first iteration which makes the algorithm to take long because pruning is not efficiently making more iterations than necessary. Finally, I used the random algorithm given in the code which provided a better initial BSSF. This is probably not the best initial BSSF we can start with but is better than start at a lower value or infinite.

6. [25] **Include a table containing the following columns.** Note that the numbers in the above table are completely made up and may or may not have any correlation with reality. Your table must include at least 10 rows of results, each for a different problem ranging between 10 and 50 cities. The first two rows should report your results on the specific cities/seeds shown above (15/20 and 16/902). Of the 10 problems, 4 must run for the full 60 seconds (before timing out and returning the best solution found so far). # of BSSF updates is the number of times a solution was found which was better than the current BSSF. A value of 0 means the final solution was just the initial BSSF. Pruned states are a) those which are created but not put on the queue because their initial bound is greater than the current BSSF, and b) any states that are put on the priority queue, but never actually expanded into children, because their lower bound became larger than the current BSSF. Just count the states actually pruned (not the many potential sub-states of those states which are also implicitly pruned).

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	1.344024	10534*	70	19	10536	10668
16	902	3.722234	7954*	80	7	25684	27977
10	82	0.024885	8036*	33	4	271	234
14	482	3.363784	6590*	58	12	27900	29251
18	187	40.802932	9887*	115	17	225213	254425
20	518	60.000726	10202	138	15	268429	301386
24	264	60.000673	11889	198	14	213647	229985
28	351	60.003442	14368	277	4	148663	169367
30	413	60.002793	13041	346	6	138253	147106
50	240	60.00960	20456	2071	5	51095	54409

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.**
As we can see from the table the algorithm has no problem in finding a cost for the TSP problem with small amount of cities before we run out of time. We can see time taken is in less than 10 seconds. This is due to the fact that number of created states is going to be smaller, and we have less iterations to make. In the algorithm we pruned states that are created but not put on the queue due to their lower bound, and those states that have a bigger lower cost than a new BSSF found. Additionally, the algorithm doesn't create a state if the conditions are not right. When

there is not path, path is infinite, or as mentioned earlier, when the cost of the calculated bound cost is bigger than the most recent BSSF found. That is why we have a total number of states created less than the ones we pruned. We can also see that for problems with lots of cities the algorithm creates big number of states since in the worst-case scenario we create n! number of states, where n is the number of cities. Because we could create and visit so many states we run out of time before we find the optimal solution.

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.**

The mechanism used in the algorithms were pruning as much and as often as it is possible and using a queue with a key that helps going deep instead of visiting all the children of a state one after the other. By visiting the children of a children so we could go deep in the tree we fin possible solutions faster then visiting a layer deeper for al children each iteration. The algorithm prunes all states that are not good to us. We remove any state has a lower cost greater than the latest BSSF found. We remove any state that has a lower cost bound of infinite. This helps reduce the number of iterations we need to make. Moreover, the key of the heap queue used for this algorithm is an important mechanism to find a solution effectively. Having the key of the priority queue be the lower cost of reducing the matrix is not good enough this would make the children have a greater lower key and thus we would visit them after we visit all parents. For that reason, we divided the lower cost of each state by the number of cities the state has visited so far this makes the key value to decrease as we visit children since the number of visited cities increases as we go deeper. As mentioned before, this helps to go down deep and find more solutions faster.