TSP Report

Priority Queue:

For the priority queue I used the heapq library which uses a heap as the priority queue which means that all of its functions should run in O(nlogn) time at worst because of how heap queues work so inserting and retrieving information is pretty cheap. The space complexity however could technically be O(n!) if we never pruned anything however because it is constantly pruned by the algorithm it never even gets close to that

Search States:

For the Search States it should have a time complexity of O(1) because I just used a dictionary so accessing information should be constant. The space complexity however will be O(n^2) because in the dictionary it holds a 2-dimensional array for the reduced cost matrix.

Reduced Cost Matrix:

The algorithm that creates and updates the reduced cost matrix has a run time of O(n^2) because it uses two for loops to check and update every item in the array. The space complexity of the algorithm is technically O(1) because the function itself doesn't create any new space but just updates the array however the greater branch and bound algorithm creates a new array every time to pass it in while not changing the parent so it is probably closer to O(n^2) for the space as well for the 2-dimensional array it creates.

BSSF Initialization:

I used the greedy algorithm for the initialization which the greedy algorithm has a time complexity of O(n^3) because it uses two for loops to run through all the cities
and create a greedy path, and it will do this n times for every city it could possibly start at
the space complexity is O(n) because all it does is create a list of all the cities in the path.

Expanding One Search State:

In order to expand one search state into its children my algorithm runs in O(n^3) time because it can create a child node for every city it hasn't been to and to create the new node it has to run the reduced cost matrix algorithm which has a time complexity of O(n^2) so overall it would be O(n^3). As mentioned before the space complexity would also be O(n^3) because it creates a new 2-dimensional array for each child, and it will create n children in the worst case so the overall worst case will be O(n^3).

Full Branch and Bound Algorithm:

The full algorithm technically has a time complexity of O(n!) because of all the expansions it could possibly make for every child that could be created however because of the pruning it will basically never actually run in that time so a better bound is O(b^n) where b is the average number of nodes put on the queue with each expansion and n is the number of cities. The space complexity is also technically O(n!) but in reality, it is also O(b^n) for the same reasons.

The data structure I used for the different states was a dictionary. This dictionary contained 4 different items. The first was how deep in the tree this node was. The second was its bound. Next was the reduced cost matrix associated with it. Finally, I kept track of the path that it had taken to get that far.

I used the heapq library to make the priority queue. The queue held tuples that had 3 elements so the priority queue could sort them. The first was just the lower bound divided by how deep it was in the tree so it would prioritize things lower in the tree. The second was a failsafe count in case two nodes had the depth and lower bound. The last thing in the structure was the dictionary data structure I described in the previous paragraph.

My approach for the initial bssf was to use the greedy algorithm I created. Essentially, it takes a city to start with and then takes the cheapest costing neighbor and goes to that next until it either hits a dead-end infinity or it creates a full working path. If it hits a dead-end then it just starts over with the next city to start with. If it creates a working path, then it just returns that working path and quits. If the greedy algorithm can't find any possible paths, then the initial bssf will be created by the default random algorithm given to us.

# of cities	Seed	Running Time	Cost of best Tour	Max states	BSSF updates	Total States	Pruned States
15	20	12.884	10534*	70	18	217893	194451
16	902	14.68	7954*	89	7	232361	209985
10	686	0.132	7049*	23	6	3298	2789
18	751	60	10478	115	10	861638	782940
18	377	60	9991	115	1	844724	768193
20	985	60	11218	146	10	778325	712004
30	27	60	14887	544	0	522067	456062
15	999	22.77	10516*	77	13	385733	340192
15	877	14.604	10024*	81	9	244244	215443
14	95	1.451	10165*	73	9	25450	22553

These numbers make sense for the most part. The lower number of cities can be run in a decent time however just adding a few nodes can make it run significantly slower. The max states also make sense because they should never be all that big but the one outlier at 544 never found any better solutions so obviously the initial bound was too high to really prune much off the queue. The variability in time complexity and pruned states makes sense because of the initial bssf. If the greedy algorithm didn't find a very good starting bound, then the branch and bound won't be able to prune very much so it will have to run a lot longer and it will make more total states.

I used a few things to get the state space search to did deeper and find solutions earlier. Mostly it came down to how I organized the priority queue. By using the bound divided by the depth, I was able to prioritize those things that were lowest in the tree first and then it prioritized those that had the lowest bound to avoid pruning everything at that depth and starting back at a higher depth. This was much better than my first idea of just adding the depth to the bound. The addition didn't make nearly enough of a change to the priority, so I had to change it to division and after that it found solutions pretty consistently even if it timed out unless there were a lot of cities.

```
# this greedy algorithm has a time complexity of O(n^3) because it uses two
for loops to run through all the cities
# and create a greedy path, and it will do this n times for every city it
could possibly start at
# the space complexity is O(n) because all it does is create a list of all
the cities in the path
   def greedy( self,time_allowance=60.0 ):
      results = {}
      cities = self._scenario.getCities()
      ncities = len(cities)
      foundTour = False
      count = 0
      bssf = None
      start_time = time.time()
      start_city = 0
      complete_path = True
      while not foundTour and time.time() - start_time < time_allowance and
start_city < ncities:
         route = []
         visited = []
         previous_edge = start_city
         route.append(cities[previous_edge])
         for i in range(ncities):
            visited.append(False)
         visited[previous_edge] = True
         for i in range(ncities - 1):
            shortest_edge = np.inf
            next_edge = 0
            found_edge = False
            for j in range(ncities):
               cost = cities[previous_edge].costTo(cities[j])
               if cost < shortest_edge and not visited[j]:
                  shortest_edge = cost
                  next_edge = j
                  found_edge = True

            if not found_edge:
               complete_path = False
            route.append(cities[next_edge])
            previous_edge = next_edge
            visited[next_edge] = True
         bssf = TSPSolution(route)
         count += 1
         if bssf.cost < np.inf and complete_path:
            # Found a valid route
            foundTour = True
         start_city += 1
      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 branch-and-bound algorithm that you
will implement
        </summary>
        <returns>results dictionary for GUI that contains three ints: cost of
best solution,
        time spent to find best solution, total number solutions found during
search (does
        not include the initial BSSF), the best solution found, and three more
ints:
        max queue size, total number of states created, and number of pruned
states.</returns>
    '''

# the time complexity of the branchAndBound is technically O(n!) however
because of the pruning
# it will basically never actually run in that time so a better bound is
O(b^n) where b is the
# average number of nodes put on the queue with each expansion and n is the
number of cities
# the space complexity is also technically O(n!) but in reality it is also
O(b^n) for the same reasons
    def branchAndBound( self, time_allowance=60.0 ):
        cities = self._scenario.getCities()
        ncities = len(cities)
        queue = []
        start_city = 0
        matrix = np.zeros((ncities, ncities))

        start_time = time.time()

        for i in range(ncities):
            for j in range(ncities):
                matrix[i, j] = cities[i].costTo(cities[j])

        bssf = self.greedy()
        if bssf['cost'] == np.inf:
            bssf = self.defaultRandomTour()
        cost = bssf['cost']
        bssf['count'] = 0
        bssf['max'] = 0
        bssf['total'] = 0
        bssf['pruned'] = 0

        matrix, bound = self.reduced_cost_matrix(matrix, ncities, 0)
        depth = 1
        queuedItem = {'depth': depth, 'bound': bound, 'matrix': matrix,
'route': np.array([])}
        heapq.heappush(queue, (bound / depth, bssf['total'], queuedItem))

        while time.time() - start_time < time_allowance and start_city <
ncities:
            if len(queue) == 0:
                heapq.heappush(queue, (bound / depth, bssf['total'], queuedItem))
                start_city += 1
```

```
        if start_city == ncities:
            break

        curItem = heapq.heappop(queue)[2]

        if curItem['bound'] >= cost:
            bssf['pruned'] += 1
            continue

        if len(curItem['route']) == ncities:
            bssf['cost'] = curItem['bound']
            cost = bssf['cost']
            bssf['count'] += 1
            full_route = []
            for i in curItem['route']:
                full_route.append(cities[int(i)])
            bssf['soln'] = TSPSolution(full_route)
            continue

        if len(curItem['route']) > 0:
            previous = int(curItem['route'][len(curItem['route']) - 1])
        else:
            previous = start_city

        for i in range(ncities):
            if i == previous:
                continue

            visited = False
            for j in curItem['route']:
                if int(j) == i:
                    visited = True
                    break

            if visited:
                continue

            curMatrix = np.copy(curItem['matrix'])
            curMatrix, curBound = self.partial_path(curMatrix, previous, i,
ncities, curItem['bound'])

            bssf['total'] += 1
            if curBound >= cost:
                bssf['pruned'] += 1
                continue

            route = curItem['route']
            route = np.append(route, i)

            newItem = {'depth': curItem['depth'] + 1, 'bound': curBound,
'matrix': curMatrix, 'route': route}
            heapq.heappush(queue, (curBound / (curItem['depth'] + 1),
bssf['total'], newItem))

        if len(queue) > bssf['max']:
            bssf['max'] = len(queue)
```

```
      bssf['pruned'] += len(queue)
      end_time = time.time()
      bssf['time'] = end_time - start_time
      return bssf

# this function runs in O(n^2) time because it runs two for loops the space
complexity is O(1) because it doesn't
# create any new space
   def reduced_cost_matrix(self, matrix, ncities, bound):
      curBound = bound
      for i in range(ncities):
         min_num = matrix[i, :].min()
         if min_num == np.inf:
            continue
         if min_num != 0:
            curBound += min_num
            for j in range(ncities):
               matrix[i, j] -= min_num

      for i in range(ncities):
         min_num = matrix[:, i].min()
         if min_num == np.inf:
            continue
         if min_num != 0:
            curBound += min_num
            for j in range(ncities):
               matrix[j, i] -= min_num

      return matrix, curBound

# this function runs in O(n^2) time because it has to run the
reduced_cost_matrix which has that
# time complexity it also has a space complexity of O(1) because it doesn't
create any new space
   def partial_path(self, matrix, row, column, ncities, bound):
      curBound = bound
      curBound += matrix[row, column]
      for i in range(ncities):
         matrix[row, i] = np.inf
         matrix[i, column] = np.inf

      matrix[column, row] = np.inf

      matrix, reduced_bound = self.reduced_cost_matrix(matrix, ncities,
curBound)

      return matrix, reduced_bound
```