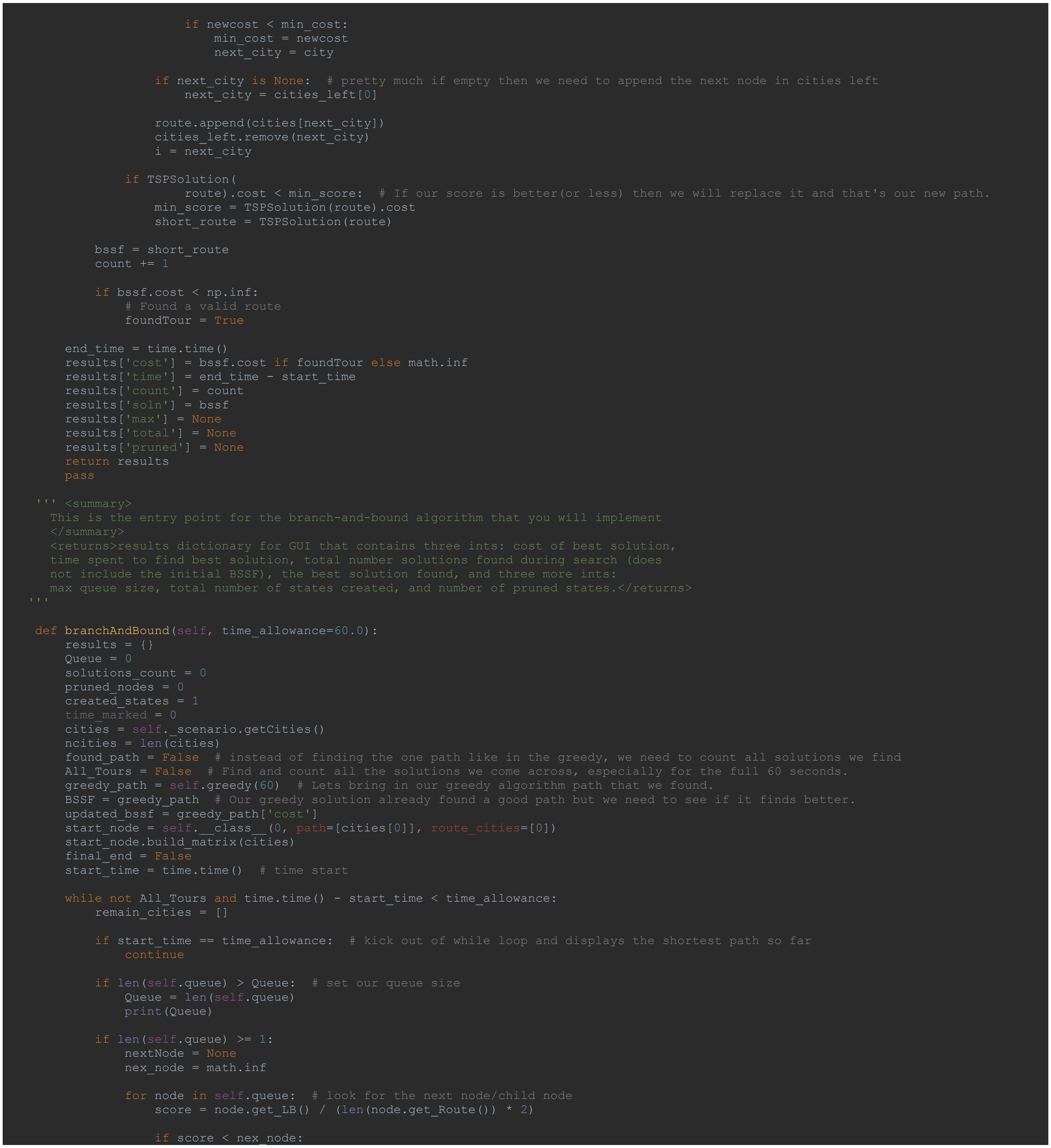
Traveling Salesperson Problem

Project 5

Tanner Webb

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

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

from which_pyqt import PYQT_VER

if PYQT_VER == 'PYQT5':
    pass
elif PYQT_VER == 'PYQT4':
    pass
else:
    raise Exception('Unsupported Version of PyQt: {}'.format(PYQT_VER))

from TSPClasses import *
import math
import copy


###### Time Complexity#####

# Sources:
# https://medium.com/analytics-vidhya/are-you-read-for-solving-the-traveling-salesman-problem-80e3c4ea45fc
# Partial Path B&B = O(n^2 * n!)
# Number of node expansions = O(b^n)
# possible start vertices = O(n)
# O(2^n) subgraph's
# Priority Queue = O(2^n) since we have visited and unvisited cities. then we also run in O(n)
# Searchstates = O(n * 2^n) for subproblems
# Reduced Cost Matrix = O(n^2)
# BSSF Initialization = O(n^2 * log2n) since I use my greedy algorithm to try and reduce the time of branch and bound
# Expanding one SearchState into others = O(n!) Since we will
# full Branch and Bound algorithm = O((n-1)!) Because we will be using factorial
# O(n^2 * b^n) for number of nodes created for potential insert when going through reduced matrix

###### Space Complexity ######

# O(n^2 * b^n) for the reduced cost matrix
# O(n^2) for each city(node) in our queue
# frontier of search tree O(b^n)

class TSPSolver:
    def __init__(self, position, path=None, cost=0, lower_bound=0, matrix_cost=None, route_cities=None):
        self._scenario = None
        if route_cities is None:
            route_cities = []
        if matrix_cost is None:
            matrix_cost = []
        if path is None:
            path = []
        self._scenario = None
        self.maxHeap = 0
        self.bssf = None
        self.queue = []  # will hold our
        self.path = path
        self.cost = cost
        self.lower_bound = lower_bound
        self.matrix_cost = matrix_cost
        self.position = position
        self.rout_cities = route_cities

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

    def get_Route(self):
        return self.path

    def path_cities(self):
        return self.rout_cities

    def get_Cost(self):
        return self.cost

    def get_LB(self):
        return self.lower_bound

    ''' <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>
   '''

    # I actually implemented this without my sources originally. I used it to try and get my shortest path and had no idea I was doing a
greedy approach
    def greedy(self, time_allowance=60.0):
        results = {}
        maxQueue = 0
        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:
            short_route = None
            min_score = math.inf

            for i in range(ncities):
                route = []
                cities_left = list(range(0, ncities))
                route.append(cities[i])
                cities_left.remove(i)

                while len(route) != ncities:
                    min_cost = math.inf
                    next_city = None

                    for city in cities_left:  # whats left of the cities, we need to check the cst with our min cost
                        newcost = cities[i].costTo(cities[city])
```



```
                        if newcost < min_cost:
                            min_cost = newcost
                            next_city = city

                    if next_city is None:  # pretty much if empty then we need to append the next node in cities left
                        next_city = cities_left[0]

                    route.append(cities[next_city])
                    cities_left.remove(next_city)
                    i = next_city

                if TSPSolution(
                        route).cost < min_score:  # If our score is better(or less) then we will replace it and that's our new path.
                    min_score = TSPSolution(route).cost
                    short_route = TSPSolution(route)

            bssf = short_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
        pass

    ''' <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):
        results = {}
        Queue = 0
        solutions_count = 0
        pruned_nodes = 0
        created_states = 1
        time_marked = 0
        cities = self._scenario.getCities()
        ncities = len(cities)
        found_path = False  # instead of finding the one path like in the greedy, we need to count all solutions we find
        All_Tours = False  # Find and count all the solutions we come across, especially for the full 60 seconds.
        greedy_path = self.greedy(60)  # Lets bring in our greedy algorithm path that we found.
        BSSF = greedy_path  # Our greedy solution already found a good path but we need to see if it finds better.
        updated_bssf = greedy_path['cost']
        start_node = self.__class__(0, path=[cities[0]], route_cities=[0])
        start_node.build_matrix(cities)
        final_end = False
        start_time = time.time()  # time start

        while not All_Tours and time.time() - start_time < time_allowance:
            remain_cities = []

            if start_time == time_allowance:  # kick out of while loop and displays the shortest path so far
                continue

            if len(self.queue) > Queue:  # set our queue size
                Queue = len(self.queue)
                print(Queue)

            if len(self.queue) >= 1:
                nextNode = None
                nex_node = math.inf

                for node in self.queue:  # look for the next node/child node
                    score = node.get_LB() / (len(node.get_Route()) * 2)

                    if score < nex_node:
                        nextNode = node
                        nex_node = score

                self.queue.remove(nextNode)
                selected_node = nextNode

            else:
                selected_node = start_node
                if final_end:
                    All_Tours = True
                    continue
                final_end = True

            for i in range(ncities):
                if not i in selected_node.path_cities():
                    remain_cities.append(cities[i])

            if len(remain_cities) == 0:
                solutions_count += 1
                found_path = True
                path_total = selected_node.get_Cost()

                if path_total < updated_bssf:

                    updated_bssf = path_total
                    BSSF = TSPSolution(selected_node.get_Route())

                    for node in self.queue:
                        if node.get_LB() > updated_bssf:
                            self.queue.remove(node)
                            pruned_nodes += 1

                time_marked = time.time() - start_time

            else:

                for place in remain_cities:
                    child = selected_node.branch_node(cities.index(place), place, cities, selected_node.position)
                    created_states += 1

                    if child.get_LB() < updated_bssf and child.get_Cost() < updated_bssf:
                        self.queue.append(child)

                    else:
                        pruned_nodes += 1

        end_time = time.time() - start_time
        print(end_time- start_time)
        pruned_nodes += len(self.queue)
        results['cost'] = BSSF.cost if found_path else math.inf
        results['time'] = end_time
        results['count'] = created_states
        results['soln'] = BSSF
        results['max'] = Queue
        results['total'] = solutions_count
        results['pruned'] = pruned_nodes
        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

    def branch_node(self, location, place, cities, origin):
        # this function is to find the child node. This helps tell the the system where we are at and were we want to go next.
        # as well as for us to determine where we need to look next

        cost = self.cost + cities[origin].costTo(place)  # add distances to current
        matrix = copy.deepcopy(self.matrix_cost)  # did much research on deepcopy. https://docs.python.org/3/library/copy.html
        path = copy.deepcopy(self.path)  # big difference if don't use deepcopy
        route_cities = copy.deepcopy(self.rout_cities)
        route_cities.append(location)
        path.append(place)
```

```python
        position = matrix[self.position]  # determines where position is
        matrix[location][self.position] = math.inf

        for i in range(len(position)):  # What do we need to set to inf after processed
            position[i] = math.inf
            matrix[i][location] = math.inf

        matrix = self.matrix_change(matrix)
        child = self.__class__(location, path=path, cost=cost, lower_bound=matrix[1],
                               matrix_cost=matrix[0], route_cities=route_cities)
        return child




    def build_matrix(self,cities):  # here we are going to make a our start matrix and retrieve the values for our cost and lb
        nCities = len(cities)

        for i in range(nCities):
            self.matrix_cost.append([])  # empty set
            row = self.matrix_cost[i]

            for j in range(nCities):
                row.append(cities[i].costTo(cities[j]))

        matrix_data = self.matrix_change(self.matrix_cost)
        self.matrix_cost = matrix_data[0]
        self.lower_bound = matrix_data[1]  # smallest element within the set




    def matrix_change(self, matrix):
        change_bound = copy.deepcopy(self.lower_bound)

        for matrix_row in matrix:
            min_row = math.inf
            for r in matrix_row:
                if r < min_row:
                    min_row = r

            k = 0
            if min_row != math.inf:
                change_bound += min_row
                for r in matrix_row:
                    matrix_row[k] = r - min_row
                    k += 1

        for matrix_column in range(len(matrix)):
            min_col = math.inf

            for c in range(len(matrix)):

                if matrix[c][matrix_column] < min_col:
                    min_col = matrix[c][matrix_column]

            if min_col != math.inf:
                change_bound += min_col

                for c in range(len(matrix)):
                    matrix[c][matrix_column] = matrix[c][matrix_column] - min_col

            # This is to see how the matrix is reducing

            # print(matrix)
            # print("----------------------------------------------------------------------------------------------------------------------")

        return [matrix, change_bound]


''' I attempted my own solution for this project and could not figure out the logic very well.
 I looked at many different resources and many different implementations on how they built their branch
 and bound but couldn't figure it out.
 https://github.com/btc36/cs312proj6/blob/2d48811b053a136edee581873b2269d2050b5fb9/TSPSolver.py#L134
 The link above is my citation to where I got my logic from. I understand how they worked on
 their branch and bound and followed their idea but modified to my version quite a bit. this resource was the only
 one that I followed correctly. I actually even learned more about python and what new function calls and class calls
 do. I know this follows the project quite a bit so I didn't want it to seem like I copied it. I really just used it
 as a resource because we only needed to implement the branch and bound which i could of done. but this made me
 want to try out the greedy algorithm. I would of thought the greedy approach would of made my program faster but
 I am getting larger numbers then my branch and bound. What I could not get to work was my nodes and my branch
```

```
and bound to display the correct solution. sometimes my greedy approach would be better then my branch and bound
solution'''
```

*<u>[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:</u>*

###### Time Complexity#####

Sources:
https://medium.com/analytics-vidhya/are-you-read-for-solving-the-traveling-salesman-problem-80e3c4ea45fc

- **Partial Path B&B** = O(n^2 * n!)
- **Number of node expansions** = O(b^n)
- **possible start vertices** = O(n)
- **subgraph's** = O(2^n)
- **Priority Queue** = O(2^n) since we have visited and unvisited cities. then we also run in O(n)
- **Searchstates** = O(n * 2^n) for subproblems
- **Reduced Cost Matrix** = O(n^2)
- **BSSF Initialization** = O(n^2 * log2n) since I use my greedy algorithm to try and reduce the time of branch and bound
- **Expanding one SearchState into others** = O(n!) Since we will
- **full Branch and Bound algorithm** = O((n-1)!) Because we will be using factorial
- O(n^2 * b^n) for number of nodes created for potential insert when going through reduced matrix

<u>###### Space Complexity ######</u>

- O(n^2 * b^n) for the reduced cost matrix
- O(n^2) for each city(node) in our queue
- frontier of search tree O(b^n)

<u>[5] Describe the data structures you use to represent the states.</u>

Some data structures we used would be:

The **priority queue** (set of cities and their values)

- Self.queue which we used to hold the queue of nodes.

Our **matrix** which is reduced and hold the values of our distances.

- Self.cost which will hold the total cost of our path.

Our **stack** which we append and pop cities to get the optimal path.

- Self.path which hold the path we want and are analyzing.

You may notice all of them used in my __init__ function. You will see my path, matrix_cost, and route_cities

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

The priority queue in my program will hold my queue which is my complete path of the optimal solution my branch and bound function finds. Originally the priority queue is supposed to find the optimal solution before time runs out. I learned that to use a priority function we needed to not only create a bounded function, but to also build a search function. Otherwise, my program may never find the optimal path and run out of time to find the optimal path. My code is not exactly the most organized, I worked hard on my priority queue to make sense.

*I spent hours trying to find my bug but somewhere along the way I changed a value or switched values so now my function does not find the optimal path above a size 8. I am still going to dig into my code and look to see if I can find the solution because I think this is a very knowledgeable project and has helped me learn so much about python.

Source:

https://realpython.com/binary-search-python/

Lower bound for sorting: Dasgupta, Sanjoy; Papadimitriou, Christos; Vazirani, Umesh. Algorithms (p. 52).

Branch and Bound: Dasgupta, Sanjoy; Papadimitriou, Christos; Vazirani, Umesh. Algorithms (p. 275).

[5] Describe your approach for the initial BSSF.

The best solution so far would be running through our bound and branch algorithm. Our branch and bound algorithm will go through the greed approach first (very quick but not the BSSF) which is why we then run it through the branch and bound and the priority queue to see if there might be a better solution. However, this takes exponentially longer to walk through. This is convenient because if our branch and bound times out, we have at least one optimal solution. This is where the greedy algorithm in my program comes in. This seems to work for the higher the sizes if our program is slow. However, our greedy algorithm is fast and takes barely any time in attempting to find the BSSF.

However, my branch and bound function has an error. I originally was just going to work on the branch and bound but I heard that if we implement the greedy algorithm into our branch and bound then it will be faster, and results will be better. So, I attempted but I created a bug in my program. My greedy algorithm which was originally turning into my branch and bound sometimes gets a better result on cost, but it is very close to the smallest route (at least the example Paul has on the assignment debug.) I will be working on this some more because I spent much time on this project so it would be good to make it work correctly.

(Update: I learned that running the path in greedy first and then doing branch and bound will give me a better result.

[25] Include a table containing the following columns.

** Below are my best solutions in my branch and bound algorithm. Not sure if it was just my program but I could not get it to calculate after 16 in my branch and bound. Greedy had no issues though Difficulty: Hard*

# Cities	Seed	Running time (sec.)	Cost of best tour found (*=optimal)	Max # of stored states at a given time	# of BSSF updates(solutions)	Total # of states created	Total # of states pruned
10	20	8	8155	27	4645	5	3429
11	290	60	7815	37	42911	5	32103
12	645	10.78	7634	45	5228	5	5228
13	697	60	9457	66	30699	13	22450
14	225	60	10769	56	27020	4	26652
15	20	60	11351 (Using greedy)	70	21515	7	16192
16	902	60	9125	87	20244	6	16137
10	20	14.7	9910	27	14113	1	10450
12	468	60	8325	44	37017	11	27614
13	431	60	9504	68	21084	7	16446

[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.

The results in the table show that my time complexity is high along with my states pruned. I think this is because its searching for the optimal solutions still, because our solutions is high as well. To me this says the program is doing its job, however I feel like that should also mean that it is finding the results quicker. I also think the The seed and the distance between the cities could also cause the program to also take longer because it has more solutions to check and there could be very close possibilities it has to calculate. As we are noticing though that the higher we go on the size, the larger the results and time taken for BSSF.

- Run time is how long the branch and bound algorithm runs to find the optimal solution is found. I programmed this to run for 60 seconds if still searching for a solution. Then to display the time when the optimal solution was found. But each time my program runs for 60 seconds on large numbers.
- Cost of best tour is my final solution found withing the 60 seconds.
- Max # of stored states I thought deals with the max queue size because we are storing those states and then saving them within our array.
- # of BSSF is also my total states. My program updates each time we see an optimal path and update that. Along with the new path

[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 struggled trying to come up with an implementation for this project. I was going to try and implement a heappq but decided to go with the priority queue since I seen we needed to discuss it in the report. I originally tried to somehow get the min path without reducing the matrix but found out it does not work past size 6. The larger the number the cost was way off from what I should. My greedy function was the start of my branch and bound but did not realize I was working on both because I never implemented my priority queue. Somewhere along the way though I got my values wrong and feel my cost are inaccurate. I know it must deal with somewhere I am not counting my edges and might possibly be missing one so I need to go back and find out where that might be.