Taylor Whitlock

Project 5: TSP

Part 1:

```python
    def greedy( self,time_allowance=60.0 ):
        results = {}
        cities = self._scenario.getCities()
        city_set = set(cities)
        best_city = None
        bssf = None
        count = 0

        start_time = time.time()
        while len(city_set) > 0 and time.time()-start_time < time_allowance:
            visited = {}
            initial_city = city_set.pop()
            visited[initial_city] = True
            curr_city = initial_city
            route = [initial_city]
            tour_complete = False
            while not tour_complete:
                closest_city = None
                cost = None
                lowest_cost = math.inf
                for city in cities:
                    if not visited.get(city):
                        cost = curr_city.costTo(city)
                        if cost < lowest_cost:
                            closest_city = city
                            lowest_cost = cost
                if lowest_cost != math.inf and cost is not None:
                    curr_city = closest_city
                    visited[curr_city] = True
                    route.append(curr_city)
                else:
                    tour_complete = (len(route) == len(cities))
                    new_bssf = TSPSolution(route)
                    if tour_complete:
                        count += 1
                        if bssf is None or new_bssf.cost < bssf.cost:
                            bssf = new_bssf
                            best_city = initial_city
                    break
        end_time = time.time()
        results['cost'] = bssf.cost if best_city is not None else math.inf
        results['time'] = end_time - start_time
        results['count'] = count
```

```python
        results['count'] = count
        results['soln'] = bssf
        results['max'] = None
        results['total'] = None
        results['pruned'] = None
        return results

    def reduce_matrix(self, matrix, lower_bound):  # returns matrix, LB
        for c_index in range(len(matrix)):  # city index and neighbor index
            low_cost = math.inf
            for n_index in range(len(matrix[c_index])):
                if matrix[c_index][n_index] < low_cost:
                    low_cost = matrix[c_index][n_index]
                    if low_cost == 0:
                        break
            if low_cost != 0 and low_cost != math.inf:
                lower_bound += low_cost
                for n_index in range(len(matrix[c_index])):
                    matrix[c_index][n_index] -= low_cost  # should make the one with the low_cost zero

        for col in range(len(matrix[0])):  # could be any index, they should all be the same
            low_cost = math.inf
            for row in range(len(matrix)):
                if matrix[row][col] < low_cost:
                    low_cost = matrix[row][col]
                    if low_cost == 0:
                        break
            if low_cost != 0 and low_cost != math.inf:
                lower_bound += low_cost
                for row in range(len(matrix)):
                    matrix[row][col] -= low_cost
        return matrix, lower_bound

    def initial_cost_matrix(self, cities):
        matrix = []
```

TSPSolver › greedy() › while len(city_set) > 0 and tim

```python

    def initial_cost_matrix(self, cities):
        matrix = []
        lower_bound = 0
        for city in cities:
            matrix_row = []
            for neighbor in cities:  # this SHOULD get all the correct indices
                if city._name == neighbor._name:
                    matrix_row.append(math.inf)
                else:
                    matrix_row.append(city.costTo(neighbor))
            matrix.append(matrix_row)
        return self.reduce_matrix(matrix, lower_bound)

    #  This function goes through and infinities out the lowest paths, then goes through and checks to make
    # sure it's still a valid cost matrix
    def update_cost_matrix(self, matrix, source, destination, lower_bound):
        for col in range(len(matrix[0])):  # zeroes out the rows and columns
            matrix[source._index][col] = math.inf
        for row in range(len(matrix)):
            matrix[row][destination._index] = math.inf
        matrix[destination._index][source._index] = math.inf
        return self.reduce_matrix(matrix, lower_bound)

    ''' <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 = {}
        cities = self._scenario.getCities()
        matrix, lower_bound = self.initial_cost_matrix(cities)
        tiebreaker = itertools.count()
        sols_found = 0
        bssf = self.greedy(time_allowance)['soln']
        max_stored = 0
        states_created = 0
        pruned = 0
        prio_queue = []
```

```python
prio_queue = []
initial_city = cities[0]
# (cost, depth, tiebreaker, lower_bound, city, matrix, path)
heapq.heappush(prio_queue, (0, 0, next(tiebreaker), lower_bound, cities[0], matrix, [initial_city]))
current_state = None

start_time = time.time()
while len(prio_queue) > 0 and time.time()-start_time < time_allowance:
    if current_state is None:
        current_state = heapq.heappop(prio_queue)
    for i in range(len(matrix[current_state[4]._index])):
        if i != current_state[4]._index and cities[i]._name != initial_city._name:
            states_created += 1
            cost = current_state[5][current_state[4]._index][i]
            if cost + current_state[3] < bssf.cost:
                new_path = current_state[-1].copy()
                new_path.append(cities[i])
                new_matrix = deepcopy(current_state[5])
                new_matrix, new_lower_bound = self.update_cost_matrix(new_matrix, current_state[4], cities[i], cost + current_state[3])
                if new_lower_bound < bssf.cost:
                    heapq.heappush(prio_queue, (cost, len(new_path) - 1, next(tiebreaker),
                                                new_lower_bound, cities[i], new_matrix, new_path))
                if len(prio_queue) > max_stored:
                    max_stored = len(prio_queue)
            else:
                pruned += 1
    current_state = heapq.heappop(prio_queue)
    if len(current_state[-1]) == len(cities):
        new_bssf = TSPSolution(current_state[-1])
        sols_found += 1
        if new_bssf.cost <= bssf.cost:
            bssf = new_bssf

end_time = time.time()
results['cost'] = bssf.cost
results['time'] = end_time - start_time
results['count'] = sols_found
results['soln'] = bssf
results['max'] = max_stored
results['total'] = states_created
results['pruned'] = pruned
return results
```

Part 2:

I used my greedy algorithm for the initial BSSF, so the time/space analysis of that should be included here, I think.

In terms of time complexity, the cost to make the initial set of cities would just be O(n), as it would have to go through each element and add it to a set. The outermost while loop is also O(n) since it just goes through each city in the set.

Afterwards, the inner while loop is roughly O(n), but could be less since it breaks once a complete tour is found. Inside that while loop is a for loop iterating through each city and checking if it’s visited, another O(n) loop, making the grand total for time complexity O(n^3).

The space complexity is O(n) for the initial set, plus O(n) for the route, plus O(n) for the dictionary of whether the city has been visited or not, making the overall asymptotic space complexity O(n).

The initial cost for the BSSF of the Branch and Bound algorithm is the cost for greedy, since that is how I got my initial BSSF, so we start with O(n^3) time and O(n) space complexity.

The cost to reduce the RCM at any time is O(n^2), since it has to go through each row and column, determine the lowest cost there, and subtract that low cost from each element. Reducing the rows/columns separately are both technically O(2n^2) and combined are O(4n^2) but asymptotically just O(n^2). Nothing new is created in the reduction, so the space complexity I believe is just O(n^2) from the matrix that is passed in.

The initial cost matrix is simply the cost of reduction plus the initial creation, therefore O(n^2) time complexity as well as O(n^2) space complexity.

The cost of updating the matrix is the cost of reduction plus O(2n) for looping through a single row and a single column to “infinity them out,” so that future child states won’t try to return to a node that has been visited already. That results in the overall asymptotic time complexity of updating being O(n^2). Since every update copies the matrix. It adds to the space complexity by a factor of O(n^2).

For time complexity of the overall branch and bound algorithm, it is the cost of the initial matrix creation plus the cost of the initial bssf(O(n^3) + O(n^2)), plus the cost of the following:

- Insertion/deletion from the priority queue (using heapq) is O(logn)
- The outermost while loop that is looping until the priority queue is empty will be something around O(2^n) because of pruning shenanigans.
- Looping through each city to create states is O(n), making the BnB algorithm alone a total of O(n^2) so far
- Copying the path adds a factor of O(n)
- Copying the matrix adds a factor of O(n^2) time and space complexity (space complexity already accounted for in the updating portion)
- Updating adds in O(n^2), explained above.

All of these together make a total time complexity of roughly O((2^n)*(n^3)). I think the absolute worst case scenario (if NOTHING gets pruned) is O((n^n)*(n^3)), since the priority queue while loop would have to run an exponential amount of times.

The space complexity of the algorithm alone is the size of the priority queue * the size of the elements stored in it. Each element is a tuple with 4 ints, a city, the O(n^2) matrix, and the O(n) path, so each tuple is roughly O(n^2) in size, making the space complexity roughly O(n^3) if my assumption about the priority queue loop is correct.

Part 3:

For each state, I used a tuple structured like the following: (cost, depth, tiebreaker, lower_bound, city, matrix, path). Cost, depth, tiebreaker, and lower_bound were all ints, the city was a TSPClasses.City, the matrix was a 2D list of ints(the costs, really the RCM), and the path was a 1D list of cities.

Part 4:

I used heapq for my priority queue, sorted by the cost of getting to that particular state, then by depth, then by a tiebreaker. Upon reflection it likely would've been better to sort by a combination of cost and depth, so that states closer to the solution had priority. With that implementation, it would probably prune a lot more states and thus create far fewer states. It is simply a min-heap queue storage system.

Part 5:

My approach for the initial BSSF was simply to use the greedy algorithm, since it tends to be quite fast but also gets a pretty decent solution to begin with.

Part 6:

1	Scenario			Greedy		Branch-and-Bound					
2	# Cities	Seed	Difficulty	Time (sec)	Tour Length	Running Time	Cost of best tour	Max # stored states	# of BSSF updates	Total # states created	Total # states pruned
3	15	20	Hard	0.002992	10251	0.214082	9137	125	2	8451	7336
4	16	902	Hard	0.003987	10680	1.589608	8078	587	13	47321	41503
5	20	542	Hard	0.007009	15002	20.087189	11124	3760	6	532441	471879
6	25	34	Hard	0.012963	13045	60.000671	11850	25357	12	1219139	1089854
7	30	759	Hard	0.022938	17278	60.000517	12938	53567	277	496105	416764
8	35	117	Hard	0.035904	17113	60.000838	16801	35208	2	473221	411744
9	40	72	Hard	0.074325	22862	60.000397	22862	37001	0	124400	84372
10	17	151	Hard	0.004017	11270	2.019356	9362	554	12	54211	47976
11	10	482	Hard	0.000998	6318	0.015957	5217	37	3	689	531
12	12	420	Hard	0.001994	7910	0.084746	7910	52	0	3941	3271

Part 7:

I find it very interesting that with my data, the largest space required was for a graph of size 25. I assume that was because the greedy algorithm found an unusually large initial solution, so the branch and bound had to check and create more states to find increasingly better solutions. When it got to size 40, my guess is that it didn't update the bssf because it ran out of time. That probably could have been improved with a slightly different implementation of how the heap was sorted.

The time complexity definitely seemed to fit the exponential growth, though I think because of the pruning it was still able to find good solutions.

The number of states pruned seemed to be about the same proportionally for each test, no matter the problem size.