Emily Morrow
Lab 5

TSP Branch & Bound

Code

```
def branchAndBound( self, start_time, time_allowance=60.0 ):
    start_time = time.time()
    tiebreaker = count()
    cities = self._scenario.getCities()
    size = len(cities)
    adjMatrix = np.zeros(shape=(size,size))
    queue = Q.PriorityQueue()

    # find the initial best solution using the default random tour algorithm
    findBssf = self.defaultRandomTour(0, 10)

    if findBssf != None:
        bssf = findBssf['soln']
    else:
        bssf = TSPSolution(cities)
        bssf.costOfRoute = float('inf')

    pruned = 0
    stored = 0
    update = 0
    states = 0
    results = {}

    # fill the matrix with initial path weights
    for i in range(size):
        for j in range(size):
            if i != j:
                dist = cities[i].costTo(cities[j])
                adjMatrix[i][j] = dist
            else:
                adjMatrix[i][j] = float("inf")

    # reduce the matrix to get the first states lower bound
    adjMatrix, weight = self.reduceMatrix(adjMatrix)
    my_path = []
    my_path.append(0)


    # add the state to the priority queue (priority, bound, tiebreaker, matrix, path)
    queue.put((0, weight, next(tiebreaker), adjMatrix, my_path))

    # while there are still states in the queue and we haven't passed the time
allowance
    while not queue.empty() and (time.time() - start_time) < time_allowance:
        if queue.qsize() > stored:
            stored = queue.qsize()

        # get the object from the queue (by priority) and store it in a state object
        s = queue.get()
        curr_state = State(s[1], s[3], s[4], s[0])

        # only expand this state if it's bound is lower than the current bssf cost
        if curr_state.bound < bssf.costOfRoute():
            children = self.makeChildren(curr_state)
            for child in children:
```

Emily Morrow
Lab 5

```
                states += 1

                # only add child states to the priority queue if it's bound is lower
than the current bssf cost
                if child['bound'] < bssf.costOfRoute():
                    queue.put((child['priority'], child['bound'], next(tiebreaker),
child['matrix'], child['path']))
                else:
                    pruned += 1

        # if the state is a full tsp path
        if len(cities) == len(curr_state.path):
            # check if better than initial bssf
            if curr_state.bound < bssf.costOfRoute():
                update += 1
                route = []
                for i in curr_state.path:
                    route.append(cities[i])

                tbssf = TSPSolution(route)
                if tbssf.costOfRoute() < float('inf'):
                    bssf = tbssf
                print("NEW BSSF: ", bssf.costOfRoute(), " vs ", curr_state.bound)


    print("stored: ", stored)
    print("update: ", update)
    print("states: ", states)
    print("pruned: ", pruned)
    print("time: ", time.time() - start_time)
    pruned += queue.qsize()
    return self.getResults(bssf.costOfRoute(), time.time() - start_time, update, bssf)
```

```
# expands a search state
def makeChildren(self, state):
    children = []
    matrix = state.matrix
    cities = self._scenario.getCities()
    size = len(cities)
    path1 = state.path.copy()
    curr_node = path1[len(path1) - 1]
    for i in range(size):
        path = state.path.copy()
        new_len = matrix[curr_node][i]
        if new_len < float('inf'):
            # if the node where i is isn't already in the path
            # then create state that includes node i
            if i not in path:
                path.append(i)
                chld_matrix = matrix.copy()

                # put inf in the places that can no longer be reached
                chld_matrix[i][curr_node] = float('inf')
                for j in range(size):
                    chld_matrix[curr_node][j] = float('inf')
                    chld_matrix[j][i] = float('inf')

                # reduce the child matrix and get its lower bound
                chld_matrix, weight = self.reduceMatrix(chld_matrix)
                chld_bound = weight + new_len + state.bound
```

```
                chld_path = path
                # set the states priority by prioritizing nodes
                # that are lower down in the tree
                chld_priority = chld_bound / len(chld_path)

                # put together the child object and add it to children
                child = {}
                child['priority'] = chld_priority
                child['path'] = chld_path
                child['bound'] = chld_bound
                child['matrix'] = chld_matrix
                children.append(child)
    return children
```

```
# reduces rows and columns of a matrix and gets the lower bound
def reduceMatrix(self, matrix):
    weight = 0

    # get the min values from each row
    mins = np.min(matrix,axis=1)
    for i in range(np.size(matrix,0)):
        currMin = mins[i]

        # if the min isn't infinity then subtract the min
        # from all values in the row
        if currMin != float('inf'):
            weight += currMin
            for j in range(np.size(matrix,1)):
                matrix[i][j] = matrix[i][j] - currMin

    # get the min values from each column
    mins = np.min(matrix, axis=0)
    for j in range(np.size(matrix, 0)):
        currMin = mins[j]
        if currMin != float('inf'):
            weight += currMin

            # if the min isn't infinity then subtract the min
            # from all values in the column
            for i in range(np.size(matrix, 1)):
                matrix[i][j] = matrix[i][j] - currMin

    return matrix, weight
```

Time & Space Complexity

Priority Queue

The priority queues operations both run in O(logn) time because when you add an element to the heap it needs to sift up which takes O(logn) time. Similarly when we pop off the queue the top element in the heap is deleted so the heap needs to readjust which also takes O(logn) time. The space complexity of the priority queue is O(n) where n is the number of states in the queue.

Emily Morrow
Lab 5

SearchStates
In the state object there are no operations so there is no time complexity to analyze. The space taken by the object will be O(1) for priority and bound. For path it will take O(n) where n is the number of cities. The matrix will take O(n^2). This results in O(1 + 1 + n + n^2) or O(n^2) space complexity.

Updating Reduced Cost Matrix
I reduce the cost matrix in the reduceMatrix() function. If n is the number of rows / columns (# of rows = # of columns) then this function will run in n*n time in the first nested for loop and then n*n time in the second nested for loop which is $2n^2$ or O(n^2). The only new variable in the function is mins which is size n so the space complexity is O(n).

BSSF Initialization
I use the defaultRandomTour to initialize the BSSF. This algorithm essentially guesses random routes until a valid route is discovered. On average this algorithm will find a solution in n time but in the worst case scenario the algorithm will guess every possible permutation which will result in O(n!) time. The space complexity is O(n) where n is the number of cities

Expanding a SearchState
I expand the SearchState in the makeChildren() function. The outer loop will run n times and the inner loop runs n times when it gets to it but the inner loop doesn't always get called. The inner loop will be called n – 1 times in the worst case but since we ignore constants this functions runs in O(n^2). For the space we are generating at most n-1 new states which take nxn space. This results in O(n*n^2) space complexity.

Full Algorithm
The complexity of the whole TSP branch and bound algorithm is O($n^2 * b^n$) where b is the branching factor. The while loop in the algorithm that goes through the priority queue would run n! times if we didn't prune but since we prune the loop runs b^n times and since we expand the search state when we run through the loop and this takes n^2 time we get the total time complexity of O($n^2 * b^n$). For the space complexity the biggest user of space will be the priority queue. The maximum amount of states we can have in our queue is b^n and each state takes up n^2 space so the space complexity is O($n^2 * b^n$) as well.

State Data Structure

```
class State(object):
    priority = None
    bound = None
    matrix = None
    path = None

    def __init__(self, bound, matrix, path, priority):
        self.bound = bound
        self.matrix = matrix
```

```
        self.path = path
        self.priority = priority
```

The State structure that I used to keep track of the states in the algorithm included a priority, a bound, a matrix, and a path. The priority variable was what was used to prioritize the state in the priority queue. I calculated priority by doing bound divided by the length of path. This made it so the queue prioritized states that had longer paths so that the state expansions went depth first instead of bredth.
The bound held the lower bound of the state. When expanding a state into children states this helped me determine the expanded solutions lower bounds. The matrix variable held the matrix that represented the state. The path variable held a list of the cities in the order that they had been visited up to that point.

Priority Queue Data Structure

The priority queue data structure that I use is the built in python PriorityQueue. The python PriorityQueue is implemented with a binary heap queue that has insertion and extraction at O(logn) time. The heap is a binary tree where the top node has the smallest value assigned to priority. The get() function gets the top node from the heap and then the heap reorganizes so that the next smallest priority floats to the top of the heap. When an element is added to the heap it floats up the heap until it is smaller than the nodes above it and larger than the nodes below it

Approach for initial BSSF

To get the initial bssf I used the defaultRandomTour function which gives the first valid random tour that it finds. I used the cost of the solution found by defaultRandomTour to prune the initial states until I found a solution with a better cost to help with pruning.

Table Results

To In the table below you can see the results of running my algorithm with a couple different numbers of cities. I noticed when I was running tests with the different numbers of cities I often found that my algorithm found larger problems harder. When I was running tests with 20 cities I would get it to run the full 60 seconds more often than I could get tests with 10 cities to run for the full 60 seconds. I also noticed that when more cities were involved there were much more states created and stored. This makes sense because the number of states grows exponentially.

Something I found strange was the # of BSSF updates. For each test these numbers were relatively close to eachother ranging from 5-23 but for one of my tests this number randomly shot up to 216. The first path that got followed must have had a higher bound that allowed for many more paths that were all slightly lower to be followed as well, resulting in a large bssf update number.

Emily Morrow
Lab 5

Table

# Cities	Seed	Running time (sec)	Cost of best tour found	Max stored states at a given time	# of BSSF updates	Total # of states created	Total # of states pruned
10	25	0.17	7618*	30	12	1141	764
11	802	0.61	7554*	40	20	3568	2673
15	20	1.63	9503*	98	16	6509	5058
16	902	7.45	8532*	93	5	26799	22544
17	542	14.20	11199*	97	5	46189	39690
19	412	21.99	10466*	116	10	61534	53166
20	805	60	10563	149	216	188272	156351
18	637	60	11629	110	23	182105	156812
19	602	60	10766	126	22	169095	148002
20	533	60	11700	173	14	173372	145870