Cody Kesler
CS 312 – 2
Lab 5 TSP Write Up

```
# O(n^2)
def greedy( self, start_time, time_allowance=60.0 ):

    results = {}
    start_time = time.time()
    count = 1
    route = [self._scenario._cities[0]]
    visitedCities = [0]
    scenario = self._scenario

    # O(n^2) to make the matrix
    matrix = np.full([len(scenario._cities), len(scenario._cities)], math.inf)
    for city1 in range(len(scenario._cities)):
        for city2 in range(len(scenario._cities)):
            cost = scenario._cities[city1].costTo(scenario._cities[city2])
            if (cost != 0):
                matrix[city1, city2] = cost

    # O(n^2) to check all the cities for every city
    curFromCity = 0
    for city1 in range(len(scenario._cities)):
        rowMin = math.inf
        rowMinIndex = 0
        for city2 in range(len(scenario._cities)):
            if(matrix[curFromCity, city2] < rowMin and city2 not in visitedCities):
                rowMin = matrix[curFromCity, city2]
                rowMinIndex = city2

        if(len(visitedCities) < len(scenario._cities)):
            curFromCity = rowMinIndex
            visitedCities.append(rowMinIndex)
            route.append(scenario._cities[rowMinIndex])

    bssf = TSPSolution(route)

    results['cost'] = bssf.cost
    results['time'] = time.time() - start_time
    results['count'] = count
    results['soln'] = bssf
    return results




# O(n^2 + (n-1)!*(log(n) + n^3 + n^2)) = O(n^3*(n-1)!)
def branchAndBound( self, start_time, time_allowance=60.0 ):

    # O(n^2) for greedy
    bssf = self.defaultRandomTour(start_time, time_allowance)
    bssf = bssf['soln']

    results = {}
    start_time = time.time()
    queueMax = 0
    count = 1
    prunedCnt = 0
    states = 0
    queue = []
```

Cody Kesler
CS 312 – 2
Lab 5 TSP Write Up

```
    # O(n^2)
    curState = LowerBoundState()
    curState.initStarting(self._scenario)

    #O(1)
    heappush(queue, (curState.weight, curState))
    states += 1

    # Worst Case is repeat O((n-1)!)
    while (queue):
        if (time.time() - start_time) > time_allowance:
            prunedCnt += len(queue)
            break
        if (len(queue) > queueMax):
            queueMax = len(queue)

        # O(log(n))
        popped = heappop(queue)
        curState = popped[1]

        if (curState.lowerBound > bssf.cost):
            prunedCnt += 1
            continue

        # Repeat n times
        for toCity in range(curState.matrix.shape[0]):

            if (curState.matrix[curState.cityId, toCity] != math.inf):
                # O(n^2) - become O(n^3) inside for loop
                subState = LowerBoundState()
                subState.initNext(curState, toCity, self._scenario._cities[toCity])
                states += 1
                if (subState.lowerBound < bssf.cost):
                    if (len(subState.path) == subState.matrix.shape[0]):
                        route = []
                        # O(n) - Only runs when new complete solution is found.
                          Becomes O(n^2) in loop
                        for i in range(subState.matrix.shape[0]):
                           route.append(self._scenario.getCities()[subState.route[i]])

                        bssf = TSPSolution(route)
                        bssf.setPath(subState.path)
                        count += 1
                    else:
                        # O(log(n))
                        heappush(queue, (subState.weight, subState))
                else:
                        prunedCnt += 1

    results['cities'] = len(self._scenario._cities)
    results['cost'] = bssf.cost
    results['time'] = time.time() - start_time
    results['count'] = count
    results['queueMax'] = queueMax
    results['prunedCnt'] = prunedCnt
    results['path'] = bssf.path
    results['soln'] = bssf
    results['states'] = states
    return results
```

Cody Kesler
CS 312 – 2
Lab 5 TSP Write Up

```
class LowerBoundState:

    def __init__(self):
        self.route = list()
        self.path = list()
        pass

    # O(n^2)
    def initStarting(self, scenario):
        self.cityId = 0
        self.cityName = scenario._cities[0]._name
        self.route.append(0)
        self.path.append(self.cityName)
        # O(n^2)
        matrix = self.makeMatrix(scenario)
        # O(n^2)
        matrix = self.reduceMatrix(matrix, -1)
        self.matrix = matrix[0]
        self.lowerBound = self.computeLowerBound(0, 0, matrix[1])
        self.weight = self.computeWeight()

    # O(n^2)
    def initNext(self, prevLowerBoundState, curCityId, curCity):
        self.cityId = curCityId
        self.cityName = curCity._name

        self.route = deepcopy(prevLowerBoundState.route)
        self.route.append(curCityId)

        self.path = deepcopy(prevLowerBoundState.path)
        self.path.append(self.cityName)

        # O(n^2)
        prevMatrix = prevLowerBoundState.matrix.copy()
        # O(n^2)
        matrix = self.reduceMatrix(prevMatrix, prevLowerBoundState.cityId)
        self.matrix = matrix[0]
        self.lowerBound = self.computeLowerBound(
                prevLowerBoundState.lowerBound,
                prevLowerBoundState.matrix[prevLowerBoundState.cityId][self.cityId],
                matrix[1])
        self.weight = self.computeWeight()
        pass

    #O(n^2)
    def makeMatrix(self, scenario):
        matrix = np.full([len(scenario._cities), len(scenario._cities)], math.inf)
        for city1 in range(len(scenario._cities)):
            for city2 in range(len(scenario._cities)):
                cost = scenario._cities[city1].costTo(scenario._cities[city2])
                matrix[city1, city2] = cost

        return matrix
```

Cody Kesler
CS 312 – 2
Lab 5 TSP Write Up

```
    #O(n^2)
    def reduceMatrix(self, matrix, prevCityId):
        #O(n)
        if(prevCityId != -1):
            matrix[:, self.cityId] = math.inf
            matrix[prevCityId, :] = math.inf
            matrix[self.cityId, prevCityId] = math.inf

        reduction = 0

        # O(n^2)
        for row in range(matrix.shape[0]):
            mini = min(matrix[row,:])
            if(mini != math.inf):
                reduction += mini
                matrix[row,:] -= mini
        # O(n^2)
        for col in range(matrix.shape[1]):
            mini = min(matrix[:,col])
            if (mini != math.inf):
                reduction += mini
                matrix[:,col] -= mini
        return (matrix, reduction)

    def computeLowerBound(self, prevLB, prevCost, thisLower):
        return prevLB + prevCost + thisLower

    def computeWeight(self):
        n = self.matrix.shape[0]
        r = len(self.route)
        return (1 + ((n-r)/n)) * self.lowerBound

    def toString(self):
        print(" cityId ", self.cityId)
        print("lowerBound ", self.lowerBound)
        print("route = ", self.route)
        print("weight = ", self.weight)
        print("matrix ")
        print(self.matrix)

    def __lt__(self, other):
        return self.path[-1] < other.path[-1]
```

Cody Kesler
CS 312 – 2
Lab 5 TSP Write Up

Complexity:

Time:

The time complexity comes down to a few things: Initialization of the best starting solution, creation of substates(includes the copying of the matrices and reducing of the matrices) and how the substates are sorted on the priority queue.

Using the greedy algorithm to initialize the bssf takes n^2 time. An initial city(city 0) is generated for n^2 time and put on the queue(O(log(n))) then pops it(O(log(n))) to make it the current city. It then runs through all the cities the current city can reach creating a loop of time n. If a city is reachable the program will generate a substate for that city in n^2 time. It will check to see if the lower bound of that substate is lower than the bssf cost. If it is then it will check to see if it is a full solution. If it's a full solution it will generate the route of cities in n time. If not, it will push it on the priority queue for log(n) time.

The priority queue could potentially have all of the (n-1)! paths in it and the program runs until the queue of potential solutions is empty which could make the running of the above paragraph (n-1)! Times resulting in a O(n^3*(n-1)!). But this will be negated by the heuristic implemented.

The heuristic is based off of a percentage the depth:

Total = total number of rows in the tree
Depth = current depth
Heuristic = (1 + (total– depth)/total))*lowerbound

This heuristic will help drive deeper into the tree (by weighting deeper states to have a lower weight in the priority queue) to find a full solution a lot faster, allowing the pruning of non-necessary paths to consider, bringing the total big-O down quite a bit

Space:

The space used in this program comes from the matrices needed in each of the generated states. Each substate uses n^2 space. The most possible states that will be generated are (n-1)! paths with n cities thus n^3*(n-1)!. But again with pruning the paths using the heuristic will but the cost down a lot.

Substate Structure:

I used an object that contains: lowerbound, path, reduced matrix, and the weight. When the substate is initialized it reduces the matrix and calculates the lowerbound and the weight(heuristic)

Priority Queue:

I used the standard heapq to implement my queue. I stored the substate object and its weight(heuristic) in a tuple for the heapq to sort by the weight.

Initial BSSF:

I used a greedy algorithm. I put all the cities in a matrix and looped through all of them just taking the lowest cost to the immediate next node.

Cody Kesler
CS 312 – 2
Lab 5 TSP Write Up

Table:

Mode	# of Cities	Seed	Running Time (sec)	Cost of Best Tour	Max of Queue	# of BSSF updates	Total # of states created	Total # of states pruned
HARD	15	20	6.450397015	10339*	3310	2998	42303	21647
HARD	16	902	0.772332907	7802*	100	4	4147	3596
HARD	10	542	0.44209671	7643*	312	83	4106	2410
HARD	15	150	3.801136971	9253*	554	15	22493	18775
HARD	20	968	60.00044227	9306	16460	9	266179	212382
HARD	22	240	34.38738203	11478*	691	3	123149	111364
HARD	25	951	60.00120878	15061	24023	4	241470	149665
HARD	28	853	60.00018406	14479	9595	11	190793	132231
HARD	30	603	60.00112104	15521	7902	79	171597	134467
HARD	40	830	60.00173616	16274	11639	19	100180	83773
HARD	50	628	60.00436878	18932	3018	10	83719	71787

Analysis:

The time it takes to run the program as the nodes gets larger increases factorial-ly. Thus the larger number of nodes will take longer to compute. The number of states created generally increased as it had more paths to explore and therefore had to generate more nodes.

The oddity is with 22 nodes on seed 240. The program was able to prune a large section of the tree in the beginning as it found a very low solution upfront because of the way the cities and paths were generated.

The larger trial runs (40 and 50) generated significantly less states because they had to dig into a deeper tree to look for solutions so they ran down only a few paths before the time expired and thus would be projected to keep decreasing as the number of nodes increased which is what happened with 60 nodes as it generated 68936 nodes.