Project 5 Report
Joseph Koetting
CS-312

1. Well-Commented Code

```
from copy import deepcopy


###########################################
# TSP Tree
# Stores Tree Data to solve shortest path
#
# Each Tree stores a reduced matrix, either it is initialized
# when the root is created or it is inherited from its
# parent.
#
# Each tree contains a path of cities that each
# of its children add on to when they are visited.
# When the length of the route equals the cities,
# it is designated as a possible solution
#
# It has two main methods, reduce list and visit
#
# When reduce list is called it removes the
# lowest value from each row and column
# then reduces each other element of each
# respective row and column
#
# When visit is called, it visits each of the
# edges to the other cities and returns a list
# of new children for each city visited.
###########################################
class TSPTree:

    ###########################################
    # def __init__
    # Initialize Tree
    #
    # Time Complexity: O(n^2) Calls make_list method
    # Space Complexity: O(1) init some variables
    ###########################################
    def __init__(self, cities, cost=0, path=[], my_list=None, visit_row=0):

        # init variables
        self.cities = cities
        self.cities_length = len(self.cities)
        self.cost = cost
        self.path = path
        self.path_length = len(self.path)
        self.my_list = my_list
        self.visit_row = visit_row

        # make new list is no list
        if self.my_list is None:
            self.my_list = self.make_list()

        self.my_list, self.cost = self.reduce_list(self.my_list, cost)

    ###########################################
    # def make_list
    # Initialize cost list
```

```
#
# Time Complexity: O(n^2) Biggest nest is two loops
# Space Complexity: O(n^2) Makes a 2 dimensional list
###########################################
def make_list(self):

    # makes an empty list
    my_list = [[] for _ in range(self.cities_length)]

    # fills list with distances
    for i in range(self.cities_length):
        for j in range(self.cities_length):
            my_list[i].append(self.cities[i].costTo(self.cities[j]))

    return my_list

###########################################
# def reduce_list
# Reduce List
#
# Time Complexity: O(n^2) Biggest nest is two loops
# Space Complexity: O(1) Just init a few variables
###########################################
def reduce_list(self, my_list, cost):

    # init var
    lowest_index = None

    # for each row, reduce
    for i in range(self.cities_length):
        for j in range(self.cities_length):

            # if zero we good
            if self.my_list[i][j] == 0:
                lowest_index = None
                break

            # if inf, pass
            if self.my_list[i][j] == float('inf'):
                continue

            # if lowest and not zero, set lowest
            if lowest_index is None or (my_list[i][j] != 0 and my_list[i][j] < my_list[i][lowest_index]):
                lowest_index = j

        # calc cost
        if lowest_index is not None:
            tmp_cost = my_list[i][lowest_index]
            cost += my_list[i][lowest_index]

            # re-balance row
            for k in range(self.cities_length):

                if my_list[i][k] == 0 or my_list[i][k] == float('inf'):
                    continue
                my_list[i][k] -= tmp_cost

        lowest_index = None

    # for each column, reduce
```

```
    for j in range(self.cities_length):
        for i in range(self.cities_length):

            # if found zero already pass
            if self.my_list[i][j] == 0:
                lowest_index = None
                break

            # if inf, pass
            if self.my_list[i][j] == float('inf'):
                continue

            # if lowest and not zero, set lowest
            if lowest_index is None or (my_list[i][j] != 0 and my_list[i][j] < my_list[lowest_index][j]):
                lowest_index = i

            # calc cost
        if lowest_index is not None:
            tmp_cost = my_list[lowest_index][j]
            cost += my_list[lowest_index][j]

            # re-balance row
            for k in range(self.cities_length):

                if my_list[k][j] == 0 or my_list[k][j] == float('inf'):
                    continue
                my_list[k][j] -= tmp_cost

        lowest_index = None

    return my_list, cost

###########################################
# def visit
# Create a list of children tree components
#
# Time Complexity: O(n^3) Loops through all
# cities n times, then calls the constructor
# for TSP tree which is O(n^2)
# Space Complexity: O(n^2) For each city create
# at most n-1 other children
###########################################
def visit(self):

    tree_list = []

    for j in range(self.cities_length):
        cost_to_visit = deepcopy(self.my_list[self.visit_row][j])

        if cost_to_visit == float('inf'):
            continue

        # add new city to path
        tmp_path = deepcopy(self.path)
        tmp_path.append(self.cities[j])

        # calculate new cost
        cost_to_visit += self.cost

        # reduce visited row to inf
```

```
            tmp_list = deepcopy(self.my_list)
            for i in range(self.cities_length):
                tmp_list[self.visit_row][i] = float('inf')

            # reduce visited column to inf
            for k in range(self.cities_length):
                tmp_list[k][j] = float('inf')

            tree_list.append(TSPTree(self.cities, cost_to_visit, tmp_path, tmp_list, j))

        return tree_list

############################################
# def __lt__
# Comparison method to compare objects of TSP TREE for heapq
# I chose to determine the priority by the cost of the reduced tree
# and how many cities it has visited. This means, that I will
# find a lot of leaf nodes, and I will prune a lot of trees.
#
# Time Complexity: O(1) Just a few comparison operators
# Space Complexity: O(1) Stores a few variables
############################################

    def __lt__(self, other):
        return self.cost - (self.path_length * 1250) < \
            other.cost - (other.path_length * 1250)

############################################
# def brandAndBound
# Use a branch and bound algorithm to find the shortest path circuit in a graph
#
# Time Complexity: Between worst case O(n! * n^3) and best case O(n^3)
# Because each child creates at worst n - 1 children and each of those
# children cost O(n^3) when they are visited/ called, the worst case scenario is
# scary big number. However because we are using branch and bound, my
# algorithm prunes bad paths that do not lead to a good solution. If I pruned
# every bad solution then I would only need to account for the time it takes to visit
# each child which is a small time cost respectively.
# Space Complexity: See Complexity of Priority Queue
############################################
############################################
# Initial Approach for BSSF
# I used the built in default random tour, which has a
# relatively small call cost, and found the smallest
# cost in the number of times it was called.
# Depending on the size of the problem, it is
# called n ^ 3 amount of times. Because it scales
# with the problem size, it is called a useful amount
# of times regardless of the size of the problem
# being tried to solve.
#
# Time Complexity: Teacher given Code
# Space Complexity: Teacher given Code
############################################
############################################
# Complexity of Priority Queue
# I implemented with the priority queue using heapq
# heapq is a built in priority queue that uses a heap
# structure to store all the nodes
#
```

```
# Time Complexity: O(log n) push and O(log n) pop
# Space Complexity: In worst case scenario this would be n!
# because every child could create n - 1 children.
# However because a lot of children get pruned it is more
# like n^3 or n^4, from how my algorithm prunes the heap
###########################################
def branchAndBound( self, time_allowance=60.0 ):

    results = {}
    cities = self._scenario.getCities()
    ncities = len(cities)
    foundTour = False
    max = 0
    total = 1
    solutions = 0
    pruned = 0
    cost = float('inf')
    bssf = None
    start_time = time.time()

    ###############################
    # DO DEFAULT REAL QUICK
    ###############################
    for i in range(ncities ^ 3):
        results = self.defaultRandomTour()
        if cost > results['cost']:
            cost = results['cost']


    ###############################
    # MY CODE STARTS HERE
    ###############################

    # Add initial tree to heap
    initial_tree = TSPTree(cities)
    heap = []
    heapq.heappush(heap, initial_tree)

    while len(heap) > 0 and time.time()-start_time < time_allowance:
        current_tree = heapq.heappop(heap)
        tree_list = current_tree.visit()
        total += len(tree_list)
        for i in range(len(tree_list)):

            # Found a solution
            if tree_list[i].path_length == ncities:
                # print("found solution")
                solutions += 1

                if bssf is None or cost >= tree_list[i].cost:
                    bssf = TSPSolution(tree_list[i].path)
                    cost = tree_list[i].cost
                    foundTour = True
                break

            if tree_list[i].cost <= cost:
                heapq.heappush(heap, tree_list[i])
            else:
                pruned += 1
```

```
        if len(heap) > max:
            max = len(heap)

################################
# MY CODE ENDS HERE
################################

end_time = time.time()
results['cost'] = bssf.cost if foundTour else math.inf
results['time'] = end_time - start_time
results['count'] = solutions
results['soln'] = bssf
results['max'] = max
results['total'] = total
results['pruned'] = pruned

return results
```

2. Time and Complexity Explained

Methods of TSP TREE

```
############################################
# def __init__
# Initialize Tree
#
# Time Complexity: O(n^2) Calls make_list method
# Space Complexity: O(1) init some variables
############################################
```

```
############################################
# def make_list
# Initialize cost list
#
# Time Complexity: O(n^2) Biggest nest is two loops
# Space Complexity: O(n^2) Makes a 2 dimensional list
############################################
```

```
############################################
# def reduce_list
# Reduce List
#
# Time Complexity: O(n^2) Biggest nest is two loops
# Space Complexity: O(1) Just init a few variables
############################################
```

```
############################################
# def visit
# Create a list of children tree components
#
# Time Complexity: O(n^3) Loops through all
# cities n times, then calls the constructor
# for TSP tree which is O(n^2)
# Space Complexity: O(n^2) For each city create
# at most n-1 other children
############################################
```

```
############################################
# def __lt__
# Comparison method to compare objects of TSP TREE for heapq
# I chose to determine the priority by the cost of the reduced tree
# and how many cities it has visited. This means, that I will
# find a lot of leaf nodes, and I will prune a lot of trees.
#
# Time Complexity: O(1) Just a few comparison operators
# Space Complexity: O(1) Stores a few variables
############################################
```

Methods of Branch and Bound

```
############################################
# def brandAndBound
# Use a branch and bound algorithm to find the shortest path circuit in a graph
#
```

```
# Time Complexity: Between worst case O(n! * n^3) and best case O(n^3)
# Because each child creates at worst n - 1 children and each of those
# children cost O(n^3) when they are visited/ called, the worst case scenario is
# scary big number. However because we are using branch and bound, my
# algorithm prunes bad paths that do not lead to a good solution. If I pruned
# every bad solution then I would only need to account for the time it takes to visit
# each child which is a small time cost respectively.
# Space Complexity: See Complexity of Priority Queue
############################################

############################################
# Initial Approach for BSSF
# I used the built in default random tour, which has a
# relatively small call cost, and found the smallest
# cost in the number of times it was called.
# Depending on the size of the problem, it is
# called n ^ 3 amount of times. Because it scales
# with the problem size, it is called a useful amount
# of times regardless of the size of the problem
# being tried to solve.
#
# Time Complexity: Teacher given Code
# Space Complexity: Teacher given Code
############################################

############################################
# Complexity of Priority Queue
# I implemented with the priority queue using heapq
# heapq is a built in priority queue that uses a heap
# structure to store all the nodes
#
# Time Complexity: O(log n) push and O(log n) pop
# Space Complexity: In worst case scenario this would be n!
# because every child could create n - 1 children.
# However because a lot of children get pruned it is more
# like n^3 or n^4, from how my algorithm prunes the heap
############################################
```

3. Description of Data Structures

```
##########################################
# TSP Tree
# Stores Tree Data to solve shortest path
#
# Each Tree stores a reduced matrix, either it is initialized
# when the root is created or it is inherited from its
# parent.
#
# Each tree contains a path of cities that each
# of its children add on to when they are visited.
# When the length of the route equals the cities,
# it is designated as a possible solution
#
# It has two main methods, reduce list and visit
#
# When reduce list is called it removes the
# lowest value from each row and column
# then reduces each other element of each
# respective row and column
#
# When visit is called, it visits each of the
# edges to the other cities and returns a list
# of new children for each city visited.
##########################################
```

4. Description of Priority Queue

```
###########################################
# Complexity of Priority Queue
# I implemented with the priority queue using heapq
# heapq is a built in priority queue that uses a heap
# structure to store all the nodes
#
# Time Complexity: O(log n) push and O(log n) pop
# Space Complexity: In worst case scenario this would be n!
# because every child could create n - 1 children.
# However because a lot of children get pruned it is more
# like n^3 or n^4, from how my algorithm prunes the heap
###########################################
```

5. Description of initial approach for BSSF

```
###########################################
# Initial Approach for BSSF
# I used the built in default random tour, which has a
# relatively small call cost, and found the smallest
# cost in the number of times it was called.
# Depending on the size of the problem, it is
# called n ^ 3 amount of times. Because it scales
# with the problem size, it is called a useful amount
# of times regardless of the size of the problem
# being tried to solve.
#
# Time Complexity: Teacher given Code
# Space Complexity: Teacher given Code
###########################################
```

6. Table

# of Cities	Seed	Running time (sec.)	Cost of best tour found (*=optimal)	Max # of stored states at a given time	# of BSSF updates	Total # of states created	Total # of states pruned
15	20	19.21	*10534	93	9	11473	9700
16	902	41.87	*7954	97	6	23369	20037
10	1	0.31	*9357	41	1	427	319
11	1	2.64	*10189	306	2	2790	2114
12	1	2.13	*9151	134	2	1632	1270
20	744	60	10891	751	2	12819	9558
25	744	60	12682	801	1	6663	6525
30	1	60	14667	804	1	4897	3299
35	1	60	15345	1611	1	4448	1669

7. Table Explained

First off, I would like to talk about the scenarios when the time taken would exceed 60 seconds, and it gave a partial solution. Despite the increase in the number of cities, each time it seemed to go create the same amount of nodes. That was interesting to me, because although more cities would definitely take longer the same amount of the same amount of nodes could be made within the same timeframe regardless of city size.

Each of the iterations 20 - 35 each stopped at 60 seconds due to the time limit, however when I did 20 cities, it pruned a considerably more amount of nodes than the others. I believe that this is because it was able to find a second solution. The more solutions that the algorithm finds, the easier it is to prune other nodes.

The iteration of 10, ran extremely fast. This is probably because my bssf was very close to the actual solution. That means that it could prune the tree early and focus on more optimal paths.

The most nodes created and pruned were in my 15 - 16 iteration. That is because the amount of cities was small enough for my algorithm to find a lot of leaf nodes. That way it could create a lot of nodes, and at the same time, prune the bad ones, it created and pruned a lot. The max size of the states was very small as well.

For my 11-12 iteration, It found the solution early on because my bssf was accurate and there are less solutions when there are less cities. My algorithm found a good solution fast, and pruned the rest.