Advanced Algorithms (CS 5512)
Project #5
Submitted by: Shishir Khanal
Submitted to: Dr Paul Bodily

1. **Include your well-commented code.**

```
    def fillcost(self, costmatrix, cities):
        #Fill the cost to reach nodes from one another
        #in the costmatrix
        for i in range(len(cities)):
            for j in range(len(cities)):
                costmatrix[i][j] = cities[i].costTo(cities[j])
        return costmatrix

    def reduceRow(self,costmatrix, rowind):
        #subtract the columns of the current row index
        #by the min value
        row = []
        for i in range(len(costmatrix)):
            row.append(costmatrix[rowind][i])
        minval = min(row)
        if minval == float('inf'):
            return 0
        for i in range(len(costmatrix[rowind])):
            costmatrix[rowind][i] -= minval
        return minval

    def reduceColumn(self,costmatrix, colind):
        #subtract the rows of the current col index
        #by the min value
        column = []
        for i in range(len(costmatrix)):
            column.append(costmatrix[i][colind])
        minval = min(column)
        if minval == float('inf'):
            return 0
        for i in range(len(costmatrix)):
            costmatrix[i][colind] -= minval
        return minval
```

```python
    def reduceCostMatrix(self,costmatrix):
        #iterate through the costmatrix to produce
        #reduced cost matrix and initial lower bound
        lowbound = 0
        for i in range(len(costmatrix)):
            lowbound += self.reduceRow(costmatrix, i)
        for i in range(len(costmatrix)):
            lowbound += self.reduceColumn(costmatrix, i)
        return lowbound

    def setColumninf(self,costmatrix, column, value):
        #set the particular column to a desired value
        for i in range(len(costmatrix)):
            costmatrix[i][column] = value

    def setRowinf(self,costmatrix, row, value):
        #set the particular row to a desired value
        for i in range(len(costmatrix)):
            costmatrix[row][i] = value

    def lastnode(self,bssf,lowerbound,cities,visited):
        #update the information to the best solution so far
        #when the last node is reached
        if lowerbound < bssf['cost']:
                    bssf['soln'] = []
                    for i in visited:
                        bssf['soln'].append(cities[i])
                    bssf['soln'] = TSPSolution(bssf['soln'])
                    bssf['cost'] = bssf['soln']._costOfRoute()
                    bssf['count'] += 1
        return bssf
```

```
    def branchAndBound( self, time_allowance=60.0 ):

        ''' <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>
        '''
        #initialize variables
        totalstates = 1
        prunnedstates = 0
        maximumstates = 0
        starttimer = time.time()
        cities = self._scenario.getCities()
        costmatrix = np.zeros([len(cities),len(cities)])
        #fill the matrix with cost values and generate reduced cost matrix
        costmatrix = self.fillcost(costmatrix, cities)
        lowerbound = self.reduceCostMatrix(costmatrix)

        #store the costmatrix and lowerbound in the priority queue
        priorityqueue = []
        heapq.heappush(priorityqueue, (len(cities) - 1, lowerbound, [0], costmatrix))
        #generate the initial bssf using a random tour function
        initialbssf = self.defaultRandomTour(time.time())
        bssf = {}
        bssf['cost'] = initialbssf['cost']
        bssf['soln'] = initialbssf['soln']
        bssf['count'] = 1
```

```python

        while len(priorityqueue) != 0 and (time.time() - starttimer) < 60:
            #prune the branches based on the min value of the pq
            state = heapq.heappop(priorityqueue)
            depth = len(cities) - state[0]
            lowerbound = state[1]
            visited = state[2]
            costmatrix = state[3]
            if depth == len(cities):
                bssf = self.lastnode(bssf,lowerbound,cities,visited)
                continue

            #create state for every possible path
            for i in range(1, len(cities)):
                newlowerbound = lowerbound
                #if the location is not invalid
                if costmatrix[visited[len(visited) - 1]][i] != float('inf'):
                    #copy the cost matrix
                    newcostmatrix = np.array(costmatrix)
                    newlowerbound += newcostmatrix[visited[len(visited) - 1]][i]
                    #set the current row & col index to inf
                    self.setRowinf(newcostmatrix, visited[len(visited) - 1], float('inf'))
                    self.setColumninf(newcostmatrix, i, float('inf'))
                    newcostmatrix[i][visited[len(visited) - 1]] = float('inf')
                    #evalaute the lower bound
                    newlowerbound += self.reduceCostMatrix(newcostmatrix)
                    totalstates += 1
                    #if the lowerbound is less than the bssf, push the info in the heap
                    if newlowerbound < bssf['cost']:
                        newvisited = list(visited)
                        newvisited.append(i)
                        heapq.heappush(priorityqueue, (len(cities) - depth - 1, newlowerbound, newvisited, newcostmatrix))
                        #maxstates is the maximum of the value of maxstate till now or the priority queue
                        maximumstates = max(maximumstates, len(priorityqueue))
                        continue
                    prunnedstates += 1

```

```python

        bssf['time'] = time.time() - starttimer
        bssf['max'] = maximumstates
        bssf['total'] = totalstates
        bssf['pruned'] = prunnedstates
        return bssf
```

2. **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:**

Function	**Time Complexity**	**Space Complexity**
fillcost()	O(n^2)	O(n^3)
reduceRow()	O(n)	O(n^3)
reduceColumn()	O(n)	O(n^3)
reduceCostMatrix()	O(n^2)	O(n^2)
setColumninf()	O(n)	O(n^2)

setRowinf()	O(n)	O(n^2)
lastnode()	O(n)	O(1)

Function	**Time Complexity**	**Space Complexity**
Priority Queue	O(nlog(n))	O(n^2)
Search States	Usually O(n^2 * 2^n) But 60 secs for the code	O(n^4)
Reduced Cost Matrix	O(n^3)	O(n^2)
BSSF Initialization	O(n^2)	O(n^2)
Expanding each Search States	O(n^6 * log(n))	O(n^4)
branchAndBound	Usually O(n^2 * 2^n) But 60 secs for the code	O(n^4)

3. **Describe the data structures you use to represent the states.**

   Priority Queue was used to store the cost matrices, lower bound, depth parameter and

   the size of the queue. For every new pruned state, all the parameters of the priority queue are updated. Also, best solution so far information is created as a dictionary which is used to store the smaller variables that assess the overall properties of the Branch & Bound implementation like Number of prunes, total cost, etc Finally, arrays were also used to store the minimum elements of the cost matrix for the lower bound evaluation.

4. **Describe the priority queue data structure you use and how it works.**

   Priority queue is a data structure that can store the parameters of our concern based on their priority values. The values can be pushed and popped at a time complexity of O(nlogn). The cost matrix and lower bound was stored in the priority queue. The queue also kept track of the size and the depth of the search space tree. The priority queue is used to expand a search tree by popping each city out of the queue one at a time and evaluating the cost matrices.

5. **Describe your approach for the initial BSSF.**

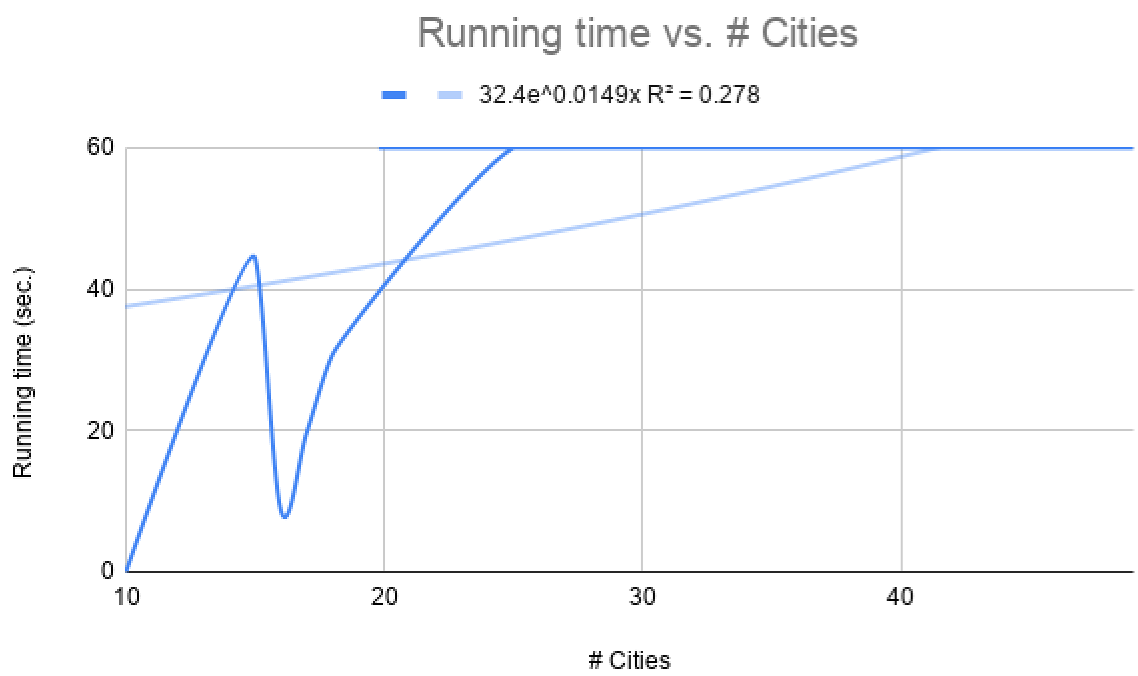
The provided code had a function called defaultRandomTour() that evaluated the parameters of bssf by using the random permutation of the cities. The function was used to create an initial values for all six parameters of the bssf dictionary

6. **Include a table containing the following columns.**

# 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
10	380	0.04	6548	29	3	231	174
15	20	44.36	10534	70	12440	189267	96204
16	902	8.6	7954	80	9	26014	22255
17	38000	19.8	11071	92	9	51926	44995
18	1100	30.8	11045	104	14	72821	62761
25	120	60	13367	224	10	83771	73355
20	11	60	11159	128	12	117416	102389
30	1800	60	19657	325	2247	76345	48933
45	180	60	20495	736	9	34110	27336
49	170	60	21556	878	6	28625	23335

7. **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 running time of the Branch & bound algorithm increased exponentially and got to the cap value really quickly as the size of the problem increased slowly. The increase in the problem size didn't show any relation to the best solutions, max states stored, number of BSSF updates, total states created and total states pruned.



8. **Discuss the mechanisms you tried and how effective they were in getting the state space search to dig deeper and find more solutions early.**

    1. **Greedy Approach:** The lowest neighbouring cost was used to perform the state space search. This had a problem with the ties creating runtime bugs in the code.

    2. **Choosing Randomly:** This approach was especially useful because this eliminated the issue of breaking the ties while choosing the next node. However, this didn't give a good solution and most of the time, the runtime was 60 secs.

    3. **PQ Push & Pop:** The values are pushed into the Priority Queue and then popped based on the min value of the cost. This got rid of the bug and the queue handled the ties automatically.