Robert Williams
Ventura – CS 213 Section 1
December 3, 2018

Traveling Salesperson Problem

Part 1 – Code

```
def branchAndBound( self, time_allowance=60.0 ):
    start_time = time.time()

    # initialization
    results = {}
    cities = self._scenario.getCities()
    n = len(cities)
    count = 0
    maxSize = 0
    totalNodes = 0
    pruned = 0
    bssf = math.inf
    bestPath = None
    # use a heap implementation of a priority queue
    pqueue = PqueueHeap()

    # build initial cost matrix -- O(n^2)
    costm = [[math.inf for _ in range(n)] for _ in range(n)]

    for i in range(n):
        for j in range(n):
            if i == j:
                continue
            costm[i][j] = cities[i].costTo(cities[j])

    lowerBound = reduceMatrix(costm)

    # get initial bssf (random)
    # average case: O(n)
    # worst case: O(n!)
    randomResult = self.defaultRandomTour()
    initialBssf = randomResult['cost']
    bssf = initialBssf

    # start with an initial branch
    branch = Branch(0, lowerBound, costm, 0, [0])

    # maximum number of branches O(n!)
    while branch != None:
        # stop if time is up
        if time.time() - start_time > time_allowance:
            break

        # prune it
        if branch.lowerBound >= bssf:
            branch = pqueue.deleteMin()
            pruned += 1
            continue

        # loop through possible child nodes - O(n^3 + log(n!))
        # there are n possible child nodes and reducing the matrix takes O(n^2)
time
        for j in range(n):
            if j == branch.city or branch.matrix[branch.city][j] == math.inf:
                continue
            totalNodes += 1

            # copy the cost matrix
            newCostm = list(map(lambda row: row[:], branch.matrix))
            newLowerBound = branch.lowerBound + newCostm[branch.city][j]

            # it is impossible to leave this city again
            newCostm[branch.city] = [math.inf for _ in range(n)]

            # it is also impossible to enter the new city again
            for i in range(n):
                newCostm[i][j] = math.inf

            # copy the path and add this city
            newPath = branch.path[:]
            newPath.append(j)

            # check if the path is complete
            if len(newPath) == n:
                # done (add the cost of returning to the start node)
                totalCost = newLowerBound + newCostm[j][newPath[0]]

                # check if this is the best solution so far
                if totalCost < bssf:
                    bssf = totalCost
                    bestPath = newPath
                    count += 1

            else:
                # don't return to the first element
                newCostm[j][newPath[0]] = math.inf

                # reduce the matrix
                newLowerBound += reduceMatrix(newCostm)

                # priority is a ratio of cost to level
                priority = newLowerBound / len(newPath)

                # prune that
                if newLowerBound >= bssf:
                    pruned += 1
                    continue

                pqueue.insert(Branch(priority, newLowerBound, newCostm, j,
newPath), priority)

                # check for max priority queue size
                if pqueue.fill > maxSize:
                    maxSize = pqueue.fill

        # get next branch from priority queue
        branch = pqueue.deleteMin()

    end_time = time.time()

    # compute route from city indeces if a better solution was found than the
initial one
    if bssf < initialBssf:
        route = list(map(lambda x: cities[x], bestPath)) if bssf < math.inf else []

    results['cost'] = bssf
    results['time'] = end_time - start_time
    results['count'] = count
    results['soln'] = TSPSolution(route) if bssf < initialBssf else
randomResult['soln']
    results['max'] = maxSize
    results['total'] = totalNodes
    results['pruned'] = pruned + pqueue.fill

    return results

class Branch:
    def __init__(self, priority, lowerBound, costMatrix, city, path):
        self.priority = priority
        self.lowerBound = lowerBound
        self.matrix = costMatrix
        self.city = city
        self.path = path

# O(n^2)
def reduceMatrix(m):
    n = len(m)
    cost = 0

    # reduce each row -- O(n^2)
    for i in range(n):
        minVal = min(m[i])

        if minVal < math.inf:
            cost += minVal
            m[i] = list(map(lambda x: x - minVal, m[i]))

    # reduce each column -- O(n^2)
    for j in range(n):
        minVal = min(list(map(lambda row: row[j], m)))

        if minVal < math.inf:
            cost += minVal
            for i in range(n):
                m[i][j] -= minVal

    return cost
```

Part 2 – Time and Space Complexity

- **Priority Queue:**
    - Time complexity:
      I used my heap priority queue implementation from the Network Routing lab. That implementation has a time complexity of O(log(k)) for inserts and O(log(k)) for popping the first item in the queue. (Where k is the number of items in the queue.) The maximum number of items in the queue is the maximum number of items at any level of the tree which is n! items. So the maximum insert and delete time is O(log(n!)).
    - Space complexity:
      The space complexity for the priority queue is O(k) because there is one entry stored for each element in the queue. (Where k is the number of items in the queue.) The maximum number of items in the queue at any time is O(n!).

- **Reducing the Cost Matrix:**
    - Time complexity:
      To reduce the cost matrix, each element of each row must be traversed (to find the minimum for that row) and each element of each column must also be traversed (to find the minimum for that column). Then the minimum for each row must be subtracted from each element in

that row and likewise for each column. So the total time complexity for reducing the cost matrix is $O(4n^2) = O(n^2)$.
  - Space complexity:
    The cost matrix has a space complexity of $O(n^2)$ for each state. (It stores the cost of a path from each city to each other city.)

- **BSSF Initialization:**
  - Time complexity:
    I use the default random path solution to find an initial BSSF. The average runtime for this solution is O(n) because usually only one solution must be tried. But it is possible that there are enough missing paths that many solutions must be tried, so the worst case running time is O(n!), which is all the permutations of cities.
  - Space complexity:
    The space complexity is O(n) because each solution is of size n and only one solution is tried at a time. If it doesn't work, it is discarded and a new solution is tried.

- **Expanding Search States:**
  - Time complexity:
    Each search state can spawn a maximum of n other search states (the number of paths from that city to other cities). The process to spawn another state includes copying the cost matrix and reducing it, and then adding it to the priority queue (if it is not pruned). So the total time complexity required to create new states from a single state is $O(n^3 + \log(n!))$, where n! Is the number of states in the priority queue.
  - Space complexity:
    When creating n new states from a single state, a new cost matrix must be created for each state. So the space complexity for this action is $O(n^3)$.

- **The Full Branch and Bound Algorithm**
  - Time complexity:
    The total time complexity of the algorithm depends on how well the "bad" states are pruned, which is not a deterministic process because the inputs are randomized. But the worst case time complexity of the algorithm requires analyzing each possible state. Since there are n! states at any given level of the tree, and there are n levels of the tree, (n+1)! total states may be considered. And the cost of considering each state is $O(n^3 + \log(n!))$, so the total time complexity must be $O((n + 1)! * (n^3 + 2*\log(n!)))$. (I multiply the log(n!) by 2 because each state must be added to the queue and also removed from the queue.
  - Space complexity:
    The total space complexity is the amount of possible states ((n + 1)! times the size of each state (n^2). So the space complexity is $O((n + 1)! * n^2)$.

Part 3 – Data Structures

I used a class called "Branch" to store information about each state. It includes the index of the current city, an array containing indeces of the cities that have already been visited (path), and a reduced cost matrix for that state (stored in a 2-dimensional array called "matrix"). The class also includes the current priority and the lower bound for that state. In order to create a new reduced cost matrix for each state, the cost matrix for the previous state has to be copied (deep copy).

Part 4 – Priority Queue

I used the priority queue implementation that I did for the Network Routing project (in which we implemented Dijkstra's algorithm). I used the heap implementation because it is relatively fast. I calculated the priority for each state as a ratio of its lower bound and its level in the tree (lowerBound/level). I found that this is a fairly efficient way to assign priority to a state because it favors branches with the lowest cost matrix while still taking into account the depth of that state in the tree.

Part 5 – Initial BSSF

I used the "defaultRandomTour" function for the initial BSSF because it is very fast and in the average case it will generate a BSSF that far from the best and worst possible paths. It is important to quickly find an initial BSSF because that will allow branches to be pruned quickly at the beginning of the algorithm.

Part 6 – Quantitative Results

# Cities	Seed	Running Time (Sec)	Cost of Best Tour (*=optimal)	Max # of Currently Stored States	# of BSSF Updates	Total # of States Created	Total # of States Pruned
15	20	0.43	9,206*	76	10	6,689	5,399
16	902	9.29	9,137*	87	6	75,839	64,693
50	483	60	22,004	1,677	4	100,413	80,142
45	213	60	18,257	1,412	8	100,900	84,536
40	87	60	18,729	1,007	9	131,104	108,081
35	621	60	14,909	749	3	163,219	132,643
30	568	60	14,809	978	7	197,136	168,206
25	943	60	13,531	218	7	244,342	213,945
20	207	37.96	9,921*	141	11	202,146	179,885
15	754	17.31	8,129*	76	16	155,995	129,101

Part 7 – Discussion of Results

I noticed that while the optimizations I make are effective, there are still times when the algorithm will take a long time simply because of a random input. For example, with 15 cities the average running time was about 1-2 seconds, but you can see on the 10th row that it took over 17 seconds to compute the shortest path with 15 cities, so there is pretty wide variability based on luck in the initial random BSSF and also in the lower bounds of different states. There is an interesting trend of more states created with smaller values of n. I attribute this to the fact that the computer can more quickly copy and reduce the matrices with smaller n values than with larger ones, so the total number of states created in 60 seconds increases as n decreases. It is also interesting to note that of all the states created, about 80% of those states were pruned (which doesn't include the states that the pruned states would have created). So the pruning allowed the algorithm to run at least 5 times faster in most cases, and probably much faster than that if you factor in the states that would have been created by the pruned states. Overall, it is clear that pruning and bounding has a significant impact on the running time of the algorithm, and it allows for a reasonable solution to be found for this very hard problem within a short amount of time.