Project 5 - Traveling Salesperson

Austin McKamey
CS312
Nov 27, 2022

1.

```
''' <summary>
     This is the entry point for the branch-and-bound algorithm that you will
implement
     </summary>
     <returns>results dictionary for GUI that contains three ints: cost of best
solution,
     time spent to find best solution, total number solutions found during search
(does
     not include the initial BSSF), the best solution found, and three more ints:
     max queue size, total number of states created, and number of pruned
states.</returns>
  '''

   def branchAndBound(self, time_allowance=60.0):  # O(n^2*logn*2^n) (time)
       start_time = time.time()
       results = self.defaultRandomTour(time_allowance)  # O(nb^n) (time)
       bssf = results['soln']
       bssf.cost = np.inf

       self.total = 1
       self.pruned = 0
       max_size = 1
       count = 0
       cities = self._scenario.getCities()
       self.ncities = len(cities)
       curr_matrix = np.zeros((self.ncities, self.ncities))

       # populate matrix with initial cost to all cities
       for i in range(self.ncities):  # O(n^2) where n is the # of cities (time)
           for j in range(self.ncities):
               curr_matrix.itemset((i, j), cities[i].costTo(cities[j]))  # O(n^2)
size of every matrix (space)
       initial_matrix, lower_bound = self.lowerBound(curr_matrix)  # O(n) (time)
       initial = Node(initial_matrix, [0])

       # priority queue is sorted by lowest depth in tree, then lower bound, and
finally node id
       heapq.heappush(self.states, (self.ncities, lower_bound, initial))  # O(logn)
(time)
                                                                           # O(b^n)
avg queue size (space)
```

```
        while len(self.states) > 0 and time.time() - start_time < time_allowance:  #
O(b^n) where b is the avg # of
                                                                                     #
nodes put on the queue expanding (time)
            tuple_p = heapq.heappop(self.states)  # O(logn) (time)
            if tuple_p[1] < bssf.cost:
                list_t = self.expand(tuple_p)  # O(n^2) (time) O(tn^2) where t is
the number of branches plus the matrix (space)
                for i in range(len(list_t)):  # O(n) where n is the # of new states
(time)
                    temp = self.test(list_t[i])
                    if temp < bssf.cost:
                        route = []
                        for j in range(len(list_t[i][2].id)):  # O(n) where n is the
# of cities (time)
                            route.append(cities[list_t[i][2].id[j]])
                        bssf = TSPSolution(route)
                        bssf.cost = temp
                        count += 1
                    elif list_t[i][1] < bssf.cost:
                        heapq.heappush(self.states, list_t[i])  # O(logn) (time)
                        if len(self.states) > max_size:
                            max_size = len(self.states)
                    else:
                        self.pruned += 1
            else:
                self.pruned += 1
        self.pruned += len(self.states)
        end_time = time.time()

        results['cost'] = bssf.cost
        results['time'] = end_time - start_time
        results['count'] = count
        results['soln'] = bssf
        results['max'] = max_size
        results['total'] = self.total
        results['pruned'] = self.pruned
        return results

    # computes the lower bound and reduced cost matrix of the state
    def lowerBound(self, cost_matrix, prev_bound=0):  # O(n) where n is the # of
cities (time & space)
        lower_bound = prev_bound
        row_min_values = np.min(cost_matrix, 1)[:, None]  # O(n) (time & space)
        for i in range(self.ncities):  # O(n) (time)
            if row_min_values[i][0] == np.inf:
```

```
                row_min_values[i][0] = 0
            lower_bound += row_min_values[i][0]  # O(1) (time & space)
        row_min_matrix = cost_matrix - row_min_values
        col_min_values = np.min(row_min_matrix, 0)[None, :]  # O(n) (time & space)
        for i in range(self.ncities):  # O(n) (time)
            if col_min_values[0][i] == np.inf:
                col_min_values[0][i] = 0
            lower_bound += col_min_values[0][i]  # O(1) (time & space)
        min_cost_matrix = row_min_matrix - col_min_values
        return min_cost_matrix, lower_bound

    # computes the valid branches coming off of the current state
    def expand(self, p):  # O(n^2) (time)
        list_t = []  # O(b) where b is the avg # of new states created (space)
        for i in range(self.ncities):  # O(n) where n is # of cities (time)
            arr = p[2].id.copy()
            first = arr[len(arr) - 1]
            # creates copy of matrix to reduce
            matrix = np.copy(p[2].cost_matrix)  # O(n^2) (space)
            prev = matrix[first, i]
            matrix[first] = np.inf
            matrix[:, i] = np.inf
            reduced_matrix, l_bound = self.lowerBound(matrix, p[1] + prev)  # O(n)
(time)
            # removing possibilities of doubling back to a previous city
            if not arr.__contains__(i):
                self.total += 1
                # pruning infinite bounds immediately
                if l_bound != np.inf:
                    arr.append(i)
                    node = Node(reduced_matrix, arr)
                    list_t.append((p[0] - 1, l_bound, node))  # O(1) (time)
                else:
                    self.pruned += 1
        return list_t

    # checks whether the tour has reached the bottom of the tree
    def test(self, t):
        if t[0] == 1:  # O(1) (time)
            return t[1]
        else:
            return np.inf

# stores the cost matrix of state and id, which is an array cities in tour
class Node:
    def __init__(self, c_m, c):
        self.cost_matrix = c_m
```

```
        self.id = c

    def __lt__(self, other):
        return self.id[0] < other.id[0]
```

2. Commented Code in #1
    a. Time Complexity
        i. branchAndBound() - $O(n^2 logn2^n)$
        ii. lowerBound() - $O(n)$
        iii. expand() - $O(n^2)$
        iv. test() - $O(1)$
    b. Space Complexity
        i. branchAndBound() - $O(tn^2b^n)$
        ii. lowerBound() - $O(n)$
        iii. expand() - $O(bn^2)$
        iv. test() - $O(1)$
3. To represent each state, I used a three-tuple consisting of the depth of the node, the lower bound of the node, and a Node object that had the cost matrix and id associated with it. The depth could also be considered height, as a leaf node actually had a "depth" of 1. The id was actually the path of cities that got to that node eg. [0,3,4].
4. I used the built in heapq structure for the priority queue, and stored the three-tuples explained in #3 on the queue. The priority key was first depth, then lower bound, and finally first value contained in the id to have an arbitrary tiebreaker in place. This allowed the algorithm to pursue tours with the greatest chance of returning a solution before backtracking to a state with a lower bound that could be much higher in the tree.
5. The initial BSSF was computed from the default random tour function given to us in the starter code.
6.

# Cities	Seed	Running Time	Cost of Tour	Max Queue Size	# Solutions	# Total States	# Pruned States
15	20	0.976	10103	76	16	9051	7597
16	902	2.116	8147	81	8	26643	23661
10	637	0.023	7316	28	2	205	166
12	547	0.244	9578	41	4	3170	2672
20	727	5.337	10163	137	21	41390	36570

25	346	60	12177	223	6	248112	210566
30	852	60	13515	325	14	194761	167186
35	252	60	18115	446	10	102123	81903
40	670	60	18187	587	2	59670	42866
45	826	60	19579	757	8	95467	82846

7. My main concern was that I was not pruning enough states, because of the discrepancy from the total states created. However, I realized that pruned states did not include those implicitly pruned, meaning that the difference should be much greater than the number of cities, which was my initial thought. As far as time, it is interesting to see how much it varies due to the initial BSSF being random. Other tests of size 20 resulted in timing out, but when I ran it for the report, it solved it in 5 seconds. This shows one of the flaws in the random approach, as well as a flaw in the algorithm. That is, it does not always get "lucky" with its choices. The biggest trend seen in the table was the max size of the queue as the size of the problems increased. Because solutions were not being found as quickly, the states were also not getting pruned, taking up space in memory, waiting to be popped off the queue.
8. The main mechanism used to find solutions early, whether they were optimal or not, was the use of depth as the primary priority key. This gave precedence to states that were closer to reaching a leaf node, and allowed every test I tried to find at least one solution. In most cases, multiple solutions were found, because of this priority. Initially, the only key I was using was lower bound, and although this guaranteed a more optimal solution, it usually timed out before it found one.