Report

CS4412

Sushan Manandhar

Project 5: TSP with Branch and Bound

Professor Bodily

15th April 2021

1. [20] Include your well-commented code.

```
def defaultRandomTour(self, time_allowance=60.0):
   results = {}
   ncities = len(self.cities)
   foundTour = False
   count = 0
   bssf = None
   start_time = time.time()
   while not foundTour and time.time() - start_time < time_allowance:
      # create a random permutation
      perm = np.random.permutation(ncities)
      route = []
      # Now build the route using the random permutation
      for i in range(ncities):
         route.append(self.cities[perm[i]])
      bssf = TSPSolution(route)
      count += 1
      if bssf.cost < np.inf:
         # Found a valid route
         foundTour = True
   end_time = time.time()
   results['cost'] = bssf.cost if foundTour else math.inf
   results['time'] = end_time - start_time
   results['count'] = count
   results['soln'] = bssf
   results['max'] = None
   results['total'] = None
   results['pruned'] = None
   return results

''' <summary>
   This is the entry point for the greedy solver, which you must
implement for
   the group project (but it is probably a good idea to just do it for
the branch-and
   bound project as a way to get your feet wet).  Note this could be
used to find your
```

```
    initial BSSF.
    </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,
the best
    solution found, and three null values for fields not used for this
    algorithm</returns>
'''


def branchAndBound(self, time_allowance=60.0):

    # initialization
    results = {}
    maxQueueSize = 0
    totalStatesCreated = 0

    prunedStatesCount = 0

    pq = []
    heapq.heapify(pq)
    start_time = time.time()
    bssf = self.greedy()['soln']
    foundTour = True
    count = 0

    # set up state
    heapq.heappush(pq, self.initFirstState())
    totalStatesCreated += 1


    while len(pq) > 0 and time.time() - start_time < time_allowance:
        currentState = heapq.heappop(pq)
        if currentState.cost < bssf.cost:
            currentState.reduce()
            if currentState.cost < bssf.cost:
                if currentState.depth == len(self.cities):
                    # update bssf
                    bssf = TSPSolution(currentState.path)
                    count += 1
                else:
                    # generate children
                    newStates = currentState.getChildren()
                    totalStatesCreated += len(newStates)
                    for state in newStates:
                        heapq.heappush(pq, state)
                    if len(pq) > maxQueueSize:
                        maxQueueSize = len(pq)
            else:
                # prune
                prunedStatesCount += 1
        else:
            # prune
            prunedStatesCount += 1

    end_time = time.time()
```

```
    results['cost'] = bssf.cost if foundTour else math.inf
    results['time'] = end_time - start_time
    results['count'] = count
    results['soln'] = bssf
    results['max'] = maxQueueSize
    results['total'] = totalStatesCreated
    results['pruned'] = prunedStatesCount
    return results

def initFirstState(self):
    cost = 0
    depth = 1
    cityIndex = 0  # start search from city 0
    path = [self.cities[cityIndex]]
    size = len(self.cities)
    table = np.empty([size, size])
    for i in range(size):
        for j in range(size):
            table[i, j] = (self.cities[i].costTo(self.cities[j]))
    initState = TSPState(cost, path, table, depth, cityIndex,
np.array(self.cities))
    initState.reduce()
    return initState

''' <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>
'''

def fancy(self, time_allowance=60.0):
    return self.greedy()



def greedy(self, time_allowance=60.0):
    results = {}
    bssf = None
    foundTour = False
    count = 0
    start_time = time.time()
    for city in self.cities:
        newPath = self._findGreedyPath(city)
        if newPath is not None:
            if bssf is None or newPath.cost < bssf.cost:
                count += 1
                bssf = newPath
                foundTour = True
        if time.time() - start_time > time_allowance:
            break
```

```
    end_time = time.time()
    results['cost'] = bssf.cost if foundTour else math.inf
    results['time'] = end_time - start_time
    results['count'] = count
    results['soln'] = bssf
    results['max'] = None
    results['total'] = None
    results['pruned'] = None
    return results

def _findGreedyPath(self, startCity):
    route = []
    visitedCities = []
    ncities = len(self.cities)
    for i in range(ncities):
        nextCity = self._getGreedyCity(startCity, visitedCities)
        if nextCity is None:
            return None
        route.append(nextCity)
        visitedCities.append(nextCity)
        startCity = nextCity
    return TSPSolution(route)

def _getGreedyCity(self, startCity, visitedCities):
    min = math.inf
    bestCity = None
    for city in self.cities:
        if city not in visitedCities:
            length = startCity.costTo(city)
            if min > length:
                min = length
                bestCity = city
    return bestCity

''' <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>
'''
```

2.[10] 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:

1. **Priority Queue:**
   - Time complexity:
     - 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!).

2. **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.)

3. **BSSF Initialization:**
   - Time complexity:
     - 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.

2. **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 $O(n^3)$.

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 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).

3. [5] Describe the data structures you use to represent the states.

I used a class called "Branch" to store information about each state. It includes the index of the current city, an array containing indices 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 must be copied (deep copy).

4. [5] Describe the priority queue data structure you use and how it works.

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 (lower Bound/level). I found that this is an 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.

4. [5] Describe your approach for the 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.

5. [25] 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
15	20	0.185054	10534	636	3	5711	4894
16	902	0.256345	7260	1152	2	1662	1399
50	483	60	22,004	1667	4	100,413	80,142
45	218	60	18,257	1412	8	100,900	84,536
40	87	60	18,729	1007	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
25	943	60	13,531	218	7	244,342	213,945

7. Results:

While the optimizations I make are successful, I have found that there are times when the algorithm will take a long time due to a random input. For e.g., the average running time for 15 cities was about 1-2 seconds, but the 10th row shows that it took over 17 seconds to calculate the results. As a result, there is a lot of variability chances in the initial random BSSF, as well as in the lower bounds of different states. More states are created with the smaller value of n than the large n values. Among all the states created, almost 80% are pruned which allowed algorithm to run at least 5 times faster in most of the cases and much faster than if the states are factored. In short, the pruning and bounding has significant impact on the run time of the algorithm and thus helps to find reasonable solution of hard problems in short period of time.