Noah Schill — Project 5: TSP

CS 312 | Fall 2021

Time and Space Complexity Analysis:

`reduce_matrix(matrix) => matrix, cost`

My reduction algorithm iterates over every row in `matrix`, calculates the minimum of each row, and subtracts that value from each value in the row (with edge cases for the infinity rows as a result of state expansion). After, the same process repeats over the transpose of `matrix` to do the same for the columns. The minimums are added up and returned as `cost` which serves as a lower-bound.

This is a subtraction which takes O(n) time, occurring n times (for each row in an n-by-n matrix) resulting in O(n^2) time complexity. This happens twice: O(2n^2) -> O(n^2)

The space complexity of this algorithm is constant as it doesn't allocate any space for a matrix or make any copies. It allocates up to 2 integers at any given point during runtime.

`travel(S: State, int: dest)`

The travel algorithm creates a copy using `np.array.copy()`. It then creates two arrays of length n (for n-cities) populated with infinity and places them at proper row and column. This takes O(n^2) time.

The space complexity is bound by the matrix copy, which is O(n^2) for n-cities.

`generate_inital_matrix()`

This creates the adjacency matrix from the list of cities. It will take O(n^2) time and space.

`greedy_bssf()`

The worst-case scenario would be O(n^3). For each state, there will be an average of n/2 neighbors left to explore, generalizing to O(n) for each state. This at worst would be done n times for n cities, resulting in O(n^3) time complexity. My implementation uses a single list that can be populated at most when a complete tour is found of n cities. Space complexity is O(n).

`branchAndBound()`

In our branch and bound function, we would need to create and search through n! if we prune nothing (our worst-case scenario). Each of these nodes take n^2 time and space bounded by `reduce_matrix()`. Thus, time complexity for `branchAndBound` is O(n!n^2).

Heap operations

- `heapq.heappop()`has to pop the root node and sift up a leaf node of a O(log n)-deep tree. Time complexity is O(log n).
- `heapq.heappush()` has to push a node to at worst the bottom of a binary heap of O(log n) depth. Time complexity is O(log n).
- `heapq.heapify()` generally takes O(n) time, but because I only use it with a list of a single node regardless of problem size, it's O(1).

The heap is stored in a python list that can contain at most n cities; the storage complexity is O(n).

Data Structures

The fundamental structure of this implementation is the `State()` class containing a reduced matrix represented as a Numpy array, a cost, and the path it has taken thus far. While this may seem redundant and easily replaceable with a tuple, Python's magic methods make it much easier to implement sorting with classes. Furthermore, it makes the code much more legible in the Branch and Bound algorithm. The BSSF in my implementation is also stored as a `State` object.

Within the algorithm itself, the heap is represented as a Python `heapq` heap. Each state object has a function `get_h()` which calculates a heuristic which prioritize states deeper in the search tree. The heuristic is then implemented within Python's magic functions which are used by `heapq` to sort a list of `State` objects into a heap.

Initialization

My first lower-bound is determined by the first cost from the row-reduction algorithm on the initial adjacency matrix.

My BSSF was implemented using a simple greedy approach.

Finding a heuristic

Initially, I was lazy and used only the cost of each State. This essentially yielded a 'greedy' approach and wasn't efficient nor optimal. Then, I tried dividing by the length of the cost. This was problematic when the initial path was zero yielding undefined priorities and was way too sensitive to the path size. In the end I opted for the following function:

$$h(S) = S.\,\mathrm{cost} - \lambda(S.\,\mathrm{PathLength})$$

I played for different values for lambda, and I found my best results with lambda = 20.

Another obstacle I ran into was the heuristic not being dynamically updated when I called it. If any parts of the State were altered, the heuristic wasn't updated. I solved this problem by making `get_h`, which computes a heuristic on the fly.

A major factor was in *when* I actually used the heuristic. I experimented with using the heuristic not only in the heap, but in my if-statements in the main `branchAndBound` while loop.

Results

# 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	0.07783	9497	490	1	655	595
16	902	0.76341	8097	4584	1	6688	5955
20	42	2.46493	9799	11534	1	16950	15521
21	947	4.911	8971	13107	1	33172	30558
25	113	60	14406	254061	0	347120	68811
30	429	60	20151	197955	0	262903	51036
34	431	60	17626	164530	0	216965	43060
42	775	60	19239	122161	0	157113	30105
50	267	60	24610	91410	0	118864	24630

I think I could've improved my heuristic by using a larger lambda value, or more heavily promoting State depth. Because of my heuristic, it prunes very aggressively on the smaller problem sizes, but tends to almost always time-out on problem sizes larger than 25. This is because the proportion of pruned states to total states decreased as my problem size increased. I think I would have had better results if I used a larger lambda, but then it would struggle with finding optimal solutions on smaller problems as the pruning might be too aggressive.

Code

```
class State:
    def __init__(self, cost, matrix: np.array, path: []):
        self.cost = cost
        self.matrix = matrix
        self.path = path

    def get_h(self):
        if self.cost != math.inf:
            return self.cost - (20 * len(self.path))
        else:
            return math.inf

    def __repr__(self):
        return f"S_{self.path}: {self.cost}"

    def __lt__(self, other):
        return self.get_h() < other.get_h()

    def __gt__(self, other):
        return self.get_h() > other.get_h()

    def __ge__(self, other):
        return self.get_h() >= other.get_h()

    def __le__(self, other):
        return self.get_h() <= other.get_h()

    def __eq__(self, other):
        return self.get_h() == other.get_h()

    def __ne__(self, other):
        return self.get_h() != other.get_h()


class TSPSolver:
    def __init__(self, gui_view):
        self._scenario = None

    def setupWithScenario(self, scenario):
        self._scenario = scenario

        def greedy_bssf(self, time_allowance=60.0):
        cost = 0
        path_of_cities = []
        path = []
        cities = self._scenario.getCities()
        current = cities[0]

        while len(path_of_cities) < len(cities):
            min_neighbor = None
            min_cost = math.inf
            for city in cities:
```

```
            neighbor_cost = current.costTo(city)
            if city not in path_of_cities and neighbor_cost < min_cost:
                min_neighbor = city
                min_cost = neighbor_cost
        current = min_neighbor
        path_of_cities.append(current)
        path.append(current._index)
        cost += min_cost

    return State(cost, None, path)  # Not returning a matrix, irrelevant

def branchAndBound(self, time_allowance=60.0):
    results = {}  # Initializing variables
    cities = self._scenario.getCities()
    n_cities = len(cities)
    start_index = 0
    start_time = time.time()
    pruned = 0
    count = 0
    total = 1  # Starting with initial node
    max_heap_size = 0

    bssf = self.greedy_bssf(cities)

    initial_matrix, initial_lb = generate_initial_matrix(
        cities)  # Generate a reduced adjacency matrix & lower bound

    s_0 = State(initial_lb, initial_matrix, [0])  # s_0 is the initial matrix

    heap = [s_0]  # Push initial value to heap
    heapq.heapify(heap)

    while len(heap) and time.time() - start_time < time_allowance:  # While our
heap is not empty
        s_n = heapq.heappop(heap)  # s_n <- heap.pop(), O(log n)
        if s_n.get_h() < bssf.get_h():  # Expand s_n
            cities_not_visited = [city for city in cities if city._index not in
s_n.path]
            for city in cities_not_visited:  # Create matrices for cities not yet
visited
                s_i = travel(s_n, city._index)  # s_i is a neighbor of s_n
                total += 1
                if s_i.cost < bssf.cost and len(s_i.path) == n_cities:  # we found
a less costly leaf node
                    bssf = s_i  # Set our best solution to be s_i
                    count += 1
                elif s_i.cost < bssf.cost:  # we found a less costly solution, but
not a leaf node
                    heapq.heappush(heap, s_i)  # O(log n)
                else:  # we found a more costly solution, time to prune
                    pruned += 1

                if len(heap) > max_heap_size:
                    max_heap_size = len(heap)
        else:
            pruned += 1
```

```
        solution_cities = []
        for city_index in bssf.path:  # Get cities from bssf list of indices
            solution_cities.append(cities[city_index])

        solution = TSPSolution(solution_cities)
        solution.cost = bssf.cost
        TSPSolver._bssf = solution
        end_time = time.time()
        results['cost'] = bssf.cost
        results['time'] = end_time - start_time
        results['soln'] = solution
        results['max'] = max_heap_size
        results['total'] = total
        results['count'] = count
        results['pruned'] = pruned
        return results


'''Static helper methods:'''
def generate_initial_matrix(cities):  # Returns lb and reduced matrix from a list of
    cities, O(n^2)
    n_cities = len(cities)
    matrix = np.empty((n_cities, n_cities))
    matrix.fill(np.inf)
    for from_index, from_city in enumerate(cities):
        for to_index, to_city in enumerate(cities):
            if from_index != to_index:
                dist = from_city.costTo(to_city)
                matrix[from_index, to_index] = dist

    return reduce_matrix(matrix)


def reduce_matrix(matrix):  # O(n^2), linear subtraction n times
    # Reduce by row
    cost = 0
    for i in range(len(matrix)):  # O(n)
        min_value = min(matrix[i])  # min of row
        if min_value != math.inf:
            matrix[i] = matrix[i] - min_value  # O(n)
            cost += min_value

    # Reduce by column
    for j in range(len(matrix)):  # O(n)
        min_value = min(matrix.T[j])  # min of col
        if min_value != math.inf:
            matrix.T[j] = matrix.T[j] - min_value
            cost += min_value

    return matrix, cost


def travel(S: State, dest: int):  # Given a state, expand to given destination. O(n)
    src = int(S.path[-1])
    new_matrix = S.matrix.copy()
    new_cost = new_matrix[src,dest]
```

```
new_matrix[src] = np.array([math.inf] * len(S.matrix))  # Infinity out the source
row
new_matrix[:, dest] = np.array([math.inf] * len(S.matrix))  # Infinity out the
destination column
new_matrix[dest, src] = math.inf  # Infinity out dest -> src

new_matrix, n = reduce_matrix(new_matrix)  # O(n)
new_cost += n + S.cost
new_path = S.path.copy()
new_path.append(dest)

return State(new_cost, new_matrix, new_path)
```