

Colin Rubow

CS 312

Project 5: TSP Project

Contents:

- 1) Well-Commented Code
- 2) Time and Space Complexity
- 3) Data Structures for States
- 4) Priority Queue
- 5) Initial BSSF Approach
- 6) Table of Trials
- 7) Discussion on Results
- 8) Search Mechanism

### 1) Well-Commented Code:

The following is the code for the BranchAndBound implementation.

```
def branchAndBound( self, time_allowance=60.0 ):
    # get set up
    cities = self._scenario.getCities() } c
    # get bssf
    greedy_result = self.greedy()
    bssf = greedy_result["cost"] O(n^3)
    num_solutions = 0
    num_pruned = 0
    num_states = 0
    # initialize array
    start_array = [[math.inf for i in range(len(cities))] for j in range(len(cities))] } O(n^2)
    for i in range(len(start_array)):
        for j in range(len(start_array[i])):
            start_array[i][j] = cities[i].costTo(cities[j]) } O(n^2)
    # start the clock
    start_time = time.time()

    # we are starting on the first city
    # set state of first city
    start_array, lower_bound = self._bound(start_array, True) } O(n^2)
    best_route = [0]
    start_state = self.State(start_array, lower_bound, best_route) } c
    pq = hpq()
    pq.insert(start_state) } O(log n)
```

```

# we are ready to start searching
while time.time() - start_time < time_allowance and pq.size() > 0:

    # expand the next state
    # get the next on the queue
    next_state_expand = pq.delete_max()  $O(\log n)$ 
    next_array = copy.deepcopy(next_state_expand.matrix)
    lower_bound = next_state_expand.lower_bound
    route_so_far = copy.deepcopy(next_state_expand.route_so_far)
    current_city = next_state_expand.get_current_city()
    # check if this state is a solution (len(route) == len(cities) + 1 && ?
    if len(route_so_far) == len(cities) + 1:
        num_solutions += 1
        # see if we have a route yet
        if best_route == [0]:
            best_route = copy.deepcopy(route_so_far)
        # check if this is new BSSF
        if lower_bound < bssf:
            bssf = lower_bound
            best_route = copy.deepcopy(route_so_far)

    # for every non inf place to go: make new state, bound new state, put
    # state on queue
    for i in range(len(cities)):
        if next_array[current_city][i] == math.inf:
            continue
        num_states += 1
        new_matrix = copy.deepcopy(next_array)
        travel_cost = new_matrix[current_city][i]
        # set row and column costs to inf
        for j in range(len(cities)):
            new_matrix[current_city][j] = math.inf
            new_matrix[j][i] = math.inf
        # set to to from indice to inf
        new_matrix[i][current_city] = math.inf
        # bound new_matrix
        new_matrix, new_lower_bound = self._bound(new_matrix, False)
        new_lower_bound = new_lower_bound + lower_bound + travel_cost
        new_route = copy.deepcopy(route_so_far)
        new_route.append(i)
        new_state = self.State(new_matrix, new_lower_bound, new_route)
        # put state on queue
        if new_state.lower_bound > bssf:
            num_pruned += 1
        else:
            pq.insert(new_state)  $O(\log n)$ 

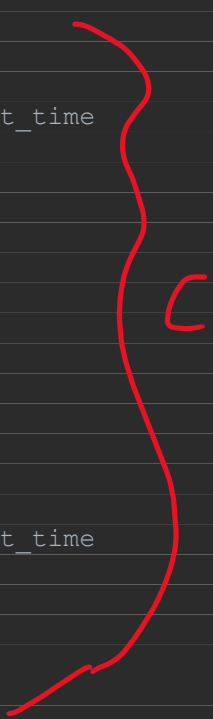
```

$O(n^2)$  (for the inner loop of the for i in range(len(cities)) block)  
 $O(n)$  (for the for j in range(len(cities)) block)  
 $O(n)$  (for the new\_matrix, new\_lower\_bound = self.\_bound(new\_matrix, False) call)  
 $O(1)$  (for the new\_lower\_bound = new\_lower\_bound + lower\_bound + travel\_cost line)  
 $O(1)$  (for the new\_route.append(i) line)  
 $O(1)$  (for the new\_state = self.State(new\_matrix, new\_lower\_bound, new\_route) line)  
 $O(1)$  (for the if new\_state.lower\_bound > bssf: line)  
 $O(1)$  (for the num\_pruned += 1 line)  
 $O(1)$  (for the else: line)  
 $O(1)$  (for the pq.insert(new\_state) line)

```

# translate indices to cities
if best_route == [0]:
    results = {}
    results["cost"] = bssf
    results["time"] = time.time() - start_time
    results["count"] = num_solutions
    results["soln"] = TSPSolution([])
    results["max"] = pq.max
    results["total"] = num_states
    results["pruned"] = num_pruned
else:
    return_route = []
    for i in best_route:
        return_route.append(cities[i])
    return_route.pop(-1)
    bssf = TSPSolution(return_route)
    results = {}
    results["cost"] = bssf.cost
    results["time"] = time.time() - start_time
    results["count"] = num_solutions
    results["soln"] = bssf
    results["max"] = pq.max
    results["total"] = num_states
    results["pruned"] = num_pruned
return results

```



## 2) Time and Space Complexity

The time complexity of this algorithm can be quite horrendous, overall, about  $O(n^n)$ . Following the code from top to bottom we see that we initialize some conditions and then perform the greedy algorithm, which has a complexity of about  $O(n^3)$ . We initialize a few more conditions, including a 2x2 array for storing the costs of going from one state to another which is clearly  $O(n^2)$ . We initialize a few more conditions and start the searching portion of our algorithm. We see a while loop and can determine that the complexity of the overall process of this loop is in  $O(n^n)$ . This is because for every city, we create  $n$  more states to explore. The next iteration we create at most  $n - 1$  more states and so forth and thus we have about  $O(n^{\frac{n}{2}})$  which is equipollent to  $O(n^n)$ . In this looping process we retrieve the next state to expand, being  $O(\log n)$  time, and get some data and make some constant time decisions. We then for every available city to go to we create a new state and place it in the queue which can be seen to be an  $O(n^3)$  process. We finish the process with some constant time reporting. The particulars of various parts of this algorithm will be discussed later.

The space complexity of this algorithm can be estimated from the maximum number of states that are stored in the priority queue. At 15 cities on Hard (Deterministic), the max queue size is at 37.

## 3) Data Structures for States

The way a state was stored was using a simple object called a State. The state held three objects. The scoring matrix, the lower bound, and the route taken thus far. It had two methods: one for retrieving the current city location, and the other for getting the length of the route taken thus far. This second method gives us an idea of how deep this state is in the branching process.

## 4) Priority Queue

In project 3 we implemented a priority queue heap. This project uses the same priority queue heap but with some tweaks. The priority was determined by the depth of the state, or how many cities the state had already visited. This guarantees that the algorithm dives deep quickly to find solutions and perform pruning quickly. The time complexity of insertion and retrieval is  $O(\log(n))$ .

## 5) Initial BSSF Approach

The initial BSSF approach uses a greedy algorithm that sequentially starts at each city and returns the best solution found in that approach.

## 6) Table of Trials

#Cities	Seed	Running Time (sec.)	Cost of Best Tour (*=optimal)	Max # of stored States	# of BSSF updates	Total # of states created	Total # states pruned
15	20	10.863958	10534*	37	3	69584	58435
16	902	60.001	8192	68	34	347024	287685
17	82	60.000	9987	73	2	317317	265335
18	453	33.698958	10197*	108	72	166378	141828
19	947	60.0016	10143	65	7	264609	228264
20	59	60.00	13716	88	10	247311	216626
21	476	60.00	0	88	0	32066	28017
22	977	60.00	0	110	0	205344	180006
23	888	60.002	0	113	0	192674	170925
24	546	60.001	0	106	0	182299	156108

## 7) Discussion on Trials

It can be seen that at higher numbers of cities the algorithm is not as likely to find a solution in the given time. The trial of 18 cities was surprising that it was able to find the optimal solution while it couldn't the previous two trials. It can be clearly seen that number of states created and overall complexity of the algorithm increases greatly with the addition of just one city.

## 8) Search Mechanism

The priority queue prioritized the depth of a state. This means the algorithm is basically performing a depth-first search for solutions. This is probably why my algorithm is not very good at finding solutions at 16+ cities. If my priority also allowed more branching than diving, maybe more solutions could be found easier and thus more pruning as well.