

Project 5 - Traveling Salesperson

In code, `Fn` refers to function complexity

In code, comments with `T O(...)` are time complexities, then comments with `S O(...)` are space complexities.

If no time complexity is provided next to a line of code, it is `O(1)` constant time.

If no space complexity is provided next to a line of code, it doesn't cost any space.

1. Source Code

node.py

Contains the node for each state within branch and bound (cost matrix, route, and lower bound)

```
from TSPClasses import TSPSolution
import numpy as np
import math
import copy

# Node containing state in the Branch and Bound Tree
class Node:
    def __init__(self, lower_bound, cost_matrix, route): # Fn - T S O(n^2)
        self.lower_bound = lower_bound
        self.cost_matrix = copy.deepcopy(cost_matrix) # T S O(n^2) - Copy items to new matrix
        self.route = copy.deepcopy(route) # T S O(n) - Copy items to new array

    def __lt__(self, other): # Fn - Practically constant - negligible comparatively
        return (self.lower_bound / len(self.route)) < (
            other.lower_bound / len(other.route)
        )

    # Returns a reduced cost matrix (0s in every row and col with the adjusted differences) for a given cost matrix
    def reduce_cost_matrix(self): # Fn - T O(n^2), S O(1)
        # Find zero and difference for each row
        row_mins = self.cost_matrix.min(axis=1) # T O(n) - loop through rows

        for i in range(len(row_mins)): # T O(n) - loop through rows
            if row_mins[i] != math.inf: # Skip over eliminated rows
                self.cost_matrix[i, :] -= row_mins[i]
                self.lower_bound += row_mins[i]

        # Find zero and difference for each col
        col_mins = self.cost_matrix.min(axis=0) # T O(n) - loop through cols

        for i in range(len(col_mins)): # T O(n) - loop through cols
            if col_mins[i] != math.inf: # Skip over eliminated cols
                self.cost_matrix[:, i] -= col_mins[i]
                self.lower_bound += col_mins[i]

    def update_child_matrix(self, row_idx, col_idx): # Fn - T O(n), S O(1)
        # Update parent bound to now have updated cost
        self.lower_bound += self.cost_matrix[row_idx, col_idx]

        # Eliminate chosen row (set to inf)
        self.cost_matrix[row_idx, :] = math.inf
```

```

# Eliminate chosen col (set to inf)
self.cost_matrix[:, col_idx] = math.inf

# Eliminate the opposite cell (don't let it go back)
self.cost_matrix[col_idx][row_idx] = math.inf

# Reduce cost matrix again
self.reduce_cost_matrix() # See Fn - T O(n^2), S O(1)

# Returns a queue of children nodes from a given parent
def expand(self, cities): # T S O((n^2)(n-1)) - n^2 to make matrix, worst case is n-1 matrices
    parent_city = self.route[-1]

    child_nodes = []

    # Get all children from parent (look through each column for parent row)
    for child_i in range(len(cities)): # T O(n - 1)
        # Skip over own cities and any ones that you cannot access from own
        if self.cost_matrix[parent_city._index, child_i] == math.inf:
            continue

        # Initialize child as parent
        child_node = Node(self.lower_bound, self.cost_matrix, self.route) # T S O(n^2)

        # Add city to route
        child_node.route.append(cities[child_i])

        # Infinity out row and col for current city and update lower bound
        child_node.update_child_matrix(parent_city._index, child_i) # T O(n), S O(1)

        child_nodes.append(child_node)

    return child_nodes

# Returns infinity if incomplete route, then returns the cost if complete
def test_complete_route(self): # Constant - practically negligible comparatively
    # Complete if it includes all cities and last has edge back to first
    if (
        len(self.route) == np.shape(self.cost_matrix)[0]
        and self.route[-1].costTo(self.route[0]) < math.inf
    ):
        return TSPSolution(self.route)
    return math.inf

```

TSPSolver.py

Contains greedy algorithm for initial BSSF and branch and bound algorithm

Greedy

```

def greedy(self, time_allowance=60.0): # Fn - T O(n^3) - Loop in a loop in a loop
                                         # Fn - S O(n) - Takes at most an array full of cities
    results = {}
    cities = self._scenario.getCities() # S O(n) - Array for all cities
    found_tour = False
    start_time = time.time()
    route = [] # S O(1) initially, but S O(n) by end (could contain all cities)

```

```

# Loop through each city
for start_city in cities: # T O(n) - Loops through all cities
    route = [start_city] # Make new route with starting city

    # For each city, try to find a route
    # T O(n) - Worst case has to do this for all the cities
    while not found_tour and time.time() - start_time < time_allowance:

        cheapest_neighbor = route[
            -1
        ] # Initialize to start city so that costTo is inf
        for neighbor in cities: # T O(n) - Loops through all cities
            if neighbor not in route and route[-1].costTo(neighbor) < route[
                -1
            ].costTo(cheapest_neighbor):
                cheapest_neighbor = neighbor

        # If invalid route - no neighbors
        # break out of while loop, move onto next city (increment i)
        if route[-1].costTo(cheapest_neighbor) == math.inf:
            break

        # Append cheapest neighbor
        route.append(cheapest_neighbor) # S O(1) - route could take up O(n)

        # If complete route (includes all cities and last has edge back to first)
        if len(route) == len(cities) and route[-1].costTo(route[0]) < math.inf:
            found_tour = True

    if found_tour:
        break

solution = TSPSolution(route)

end_time = time.time()
results["cost"] = solution.cost if found_tour else math.inf
results["time"] = end_time - start_time
results["count"] = 1 if found_tour else 0 # Greedy only finds 1
results["soln"] = solution
results["max"] = None
results["total"] = None
results["pruned"] = None
return results

```

Branch and Bound

```

def branchAndBound(self, time_allowance=60.0): # T S O((n^2)(n!))
    results = {}
    cities = self._scenario.getCities() # S O(n)
    num_cities = len(cities)

    # Make the upper bound be the greedy solution
    greedy_res = self.greedy(time_allowance)
    bssf = greedy_res["soln"] # T O(n^3), S O(n)

    # Initialize other returned variables
    solutions_count = 0 # Number of complete solutions (number of times hit bottom)
    max_queue_size = 0
    node_total = 0
    pruned_total = 0

    # Start timer

```

```

start_time = time.time()

# Make cost matrix
matrix = np.zeros((num_cities, num_cities)) # T S  $O(n^2)$ 
for i in range(np.shape(matrix)[0]): # Rows
    for j in range(np.shape(matrix)[1]): # Cols
        matrix[i][j] = cities[i].costTo(cities[j])

# Make Node class
start_node = Node(0, matrix, [cities[0]]) # T S  $O(n^2)$ 
node_total += 1

# Reduce
start_node.reduce_cost_matrix() # T  $O(n^2)$ , S  $O(1)$ 

# Make queue (following B&B pseudo code from here)
q = []
heapq.heappush(q, start_node) # T  $O(\log n)$ 

while len(q) > 0 and time.time() - start_time < time_allowance:
    if len(q) > max_queue_size:
        max_queue_size = len(q)

    cheapest_node = heapq.heappop(q) # T  $O(\log n)$ 

    if cheapest_node.lower_bound < bssf.cost:
        child_nodes = cheapest_node.expand(cities) # T S  $O((n^2)(n-1))$ 
        node_total += len(child_nodes)

        for child_node in child_nodes: # T  $O(n-1)$  - worst case # of children
            # If you hit the bottom of tree (complete route)
            if child_node.test_complete_route() < bssf.cost:
                bssf = TSPSolution(child_node.route)
                solutions_count += 1
            elif child_node.lower_bound < bssf.cost:
                heapq.heappush(q, child_node) # T  $O(\log n)$ 
            else: # Skip over it (prune)
                pruned_total += 1

end_time = time.time()
results["cost"] = bssf.cost
results["time"] = end_time - start_time
results["count"] = solutions_count
results["soln"] = bssf
results["max"] = max_queue_size
results["total"] = node_total
results["pruned"] = pruned_total
return results

```

2. Time and Space Complexities

Priority Queue

Time: $O(\log n)$

Space: $O(\log n)$

Since I used the `heapq` implementation, the time and space complexity for all `heapq` operations is $O(\log n)$, including `heappush()` and `heappop()`.

SearchStates

See source code for `Node.py` for details. The time complexities for each functions are listed below in 'Reducing Cost Matrix', 'Updating Cost Matrix', and 'Expanding one SearchState into others'

Reducing Cost Matrix

Time: $O(n^2)$

Space: $O(1)$



See `reduce_cost_matrix()`

Time complexity is n^2 because in order to reduce the cost matrix, you have to loop through each row, finding the minimum value and adjusting the row values accordingly. You must do the same for columns, so that makes it n^2 . As for space, the space for the cost matrix has already been allocated, and we're just updating values within that matrix, not creating any more variables or allocating any more memory, so the space complexity within this function is constant.

Updating Cost Matrix

Time: $O(n^2)$

Space: $O(1)$



See `update_child_matrix()` and `reduce_cost_matrix()` (called by update)

I used np array operations which have speed-ups, but assuming that under the hood there weren't any speed-ups and they functioned just like using arrays and for loops, the time complexity would be n , because for each column in the chosen row, you loop through and change the value to infinity (n operation). Then you go and do the same thing for each row in the chosen column. When you add these two operations together it results in a $O(n)$ time complexity. Then at the end of the row and column updates, you reduce which is $O(n^2)$, which dominates as the overall time complexity. As for space complexity, it is constant since we already allocated space for the matrix when we made it, and now we're just updating values within it — no new variables or allocated space is made.

Initial BSSF Initialization

Time: $O(n^3)$

Space: $O(n)$



See `greedy()`

It takes n^3 time because it has a loop within a loop within a loop as for each city, it at most has to go look through every other city iteratively. Space is just n because at worst the list would contain all cities.

Expanding one SearchState into others

Time: $O(n^2(n-1))$

Space: $O(n^2(n-1))$



See `expand()` and `update_child_matrix()` (called from `expand`)

Space complexity for a to add a node is n^2 because you'll have to make a matrix, and to do that requires a double for loop, so time complexity is also n^2 . Now you have to add nodes for all the children, and the worst case or the most amount of children is $n-1$, so the worst case scenario for time and space complexity for a single expansion is $O(n^2(n-1))$.

Full Branch and Bound Algorithm

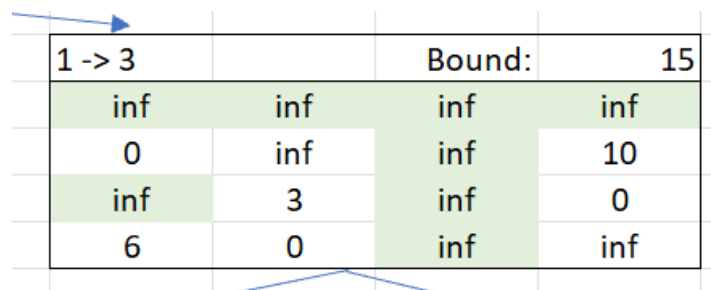
Time: $O(n^2n!)$

Space: $O(n^2n!)$

The full algorithm is n factorial for both time and space. The logic behind it is that from the starting city, it will have $(n-1)$ children (all the cities except itself). Then there will be $(n-2)$ children, and so on. Therefore, worst case scenario, there will be $n!$ children. For each state/node (the parent and all the children), they will need to have a 2D cost matrix, so that will cost n^2 . The cost of expansion (n^2) times all the children ($n!$), the overall complexity is $O(n^2n!)$. This is so much larger than n^3 complexity for greedy used for the initial bssf, so the greedy complexity can be discarded.

3. State Data Structure

For my data structure used to represent states (or nodes in my case), I defined a Python class, then made 3 data members which replicate the variables we needed to store for branch and bound at every spot in the solution.



1 -> 3		Bound:	15
inf	inf	inf	inf
0	inf	inf	10
inf	3	inf	0
6	0	inf	inf

The above picture shows conceptually how the state data structure looks like and the data it holds. It holds the 3 following data structures internally:

1. Lower bound (a Python number which represents the cost of the partial path including that state)
2. Cost matrix (a Numpy 2D array/matrix which stores numbers for the costs from every city to every other city)

3. Route (also referred to as the partial path, this is a list of cities data structure which shows the cities that are part of the solution being built)

See source code on part 1 for full source code, but here's the snippet for my State/Node class and its constructor with its class data members:

```
# Node containing state in the Branch and Bound Tree
class Node:
    def __init__(self, lower_bound, cost_matrix, route):
        self.lower_bound = lower_bound
        self.cost_matrix = copy.deepcopy(cost_matrix)
        self.route = copy.deepcopy(route)
```

4. Priority Queue Data Structure

For my priority queue data structure, I just used a simple list in Python, then used `heapq`'s `heappush()` and `heappop()` functions to control push and popping from the queue. The items in the list were state nodes. To control priority of which to pop off first, I overrode the less than operator for my state node class. See part 8 for details on that decision.

Here's a few snippets of how I used the queue (see source code on part 1 for full code):

```
# Make queue (following B&B pseudo code from here)
q = []
heapq.heappush(q, start_node) # Only have start node to begin

...
cheapest_node = heapq.heappop(q)

...
heapq.heappush(q, child_node)
```

5. Initial BSSF Approach

For my initial BSSF, at first it was infinity, but that didn't prune enough. To allow for more pruning, I switched the initial BSSF to use the default random solution, which helped quite a bit, but was still slow. I realized the best way to have a tight upper bound would be to implement the greedy algorithm and use that for my initial BSSF. Upon using greedy for the initial, the overall speed of my algorithm increased greatly.

See the source code on part 1 to see the full greedy solution code, but here's a snippet of how I took the solution from greedy and applied it here:

```
def branchAndBound(self, time_allowance=60.0):
    ...
    # Make the upper bound be the greedy solution
    greedy_res = self.greedy(time_allowance)
```

```
bssf = greedy_res["soln"] # Initially set the bssf to be greedy's
...
```

6. Results Table

# 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	4.65	*9824	66	4	2773	2219
16	902	27.51	*8160	87	7	10562	8984
10	3	0.32	*8040	10	1	322	239
12	20	3.78	*8469	35	6	3631	2645
14	1	28.89	*9525	76	9	13005	10506
20	4	60.03	10704	154	2	9219	7500
33	5	60.19	18234	444	2	2695	1506
40	8	60.09	17293	623	2	1960	998
45	9	60.03	18209	788	1	1668	583
50	10	60.13	21612	964	2	1635	434

7. Results Table Discussion

Across the board, I noticed that the larger the problem size, the longer the algorithm ran before it either found the optimal solution or timed out. This was expected, because adding more cities means a larger n on our problem which takes $O(n!)$ time and space — there's more cities to visit and paths to check. Therefore, time complexity increases substantially with problem size. As for what I noticed with pruned states, it seemed that those which ran longer had more pruned. This makes sense, because the longer the algorithm runs, the more time it has to iterate and throw away solutions as it goes. Lastly, in comparing pruned states and how they vary with problem size, there were generally more states pruned when the problem size was higher. Similar to the previous logic, it makes sense that if you have more states, meaning more possible routes, that you likewise will have more work ahead of you in terms of bad options that you can prune away.

8. Optimization Mechanisms

To help optimize my algorithm so that it would balance digging deeper down the tree and finding the cheapest states, I ended up adding complexity to my algorithm. At first my priority function (via overriding the less than operator for my state) was:


```
return self.lower_bound <= other.lower_bound
```

I realized that while this was good, this didn't account for how much deeper one state might have to go down the tree before hitting the bottom compared to another. To account for this, I changed my priority function to be (pseudo code below, see `_lt_()` in source code for actual code):

```
return (self.lower_bound + self_depth_left)
    < (other.lower_bound + other_depth_left)
```

After using this for a while, I realized I could get even better speed-up if I prioritized States/Nodes that had more cities currently in their route, so I updated my priority function to take the lower bound of the first and divide it by the number of cities in its route, then compared it to the second. This yielded the best results yet.

```
return (self.lower_bound / len(self.route)) < (
    other.lower_bound / len(other.route)
)
```