Complexity Analysis

Greedy

```
def solve(self) -> list[SolutionStats]:
    for nodeindex in range(len(self.edges)):
        if self.timer.time_out():
            return
        self.greedy_helper([], nodeindex)
    return self.stats

def greedy_helper(self, visited: list[int], currnode: int):
...
    nextnode = find_min_unvisited(self.edges[currnode], visited)

...
    return self.greedy_helper(visited, nextnode)
```

Time Complexity

The heart of my greedy algorithm is, from the current node, finding and traversing down only the minimum weight node that has not yet been visited. This causes the helper to only recuse down one path at a time from each start node. Because of this, the greedy helper function when run only has a O(n), and it is run n times. Thus, the overall time complexity of my greedy algorithm is $O(n^2)$.

Space Complexity

Since the helper function is recursive, it stores a complete list of all variables it needs on the system stack, and for every run that would be a O(n) because for the worst-case scenario (it finds a valid path) it traverses through every node in the tree once. The space complexity however is still O(n) as opposed to $O(n^2)$ because all of the data stored by greedy_helper is popped off the stack completely when a new start node is considered.

Depth First Search

```
def dfs_recursive(self, currnode: int, visited: list):
    for edge_index in range(0, len(self.edges[currnode])):
        if self.edges[currnode][edge_index] == math.inf:1
```

```
            continue

        if edge_index in visited:
            continue

        self.dfs_recursive(edge_index, visited.copy())
```

Time Complexity

My depth first search implementation for the traveling salesman problem will go down each node in order from 0 to n and see if it can continue to traverse. If it can, then it will recurse and set the next node to currnode and continue. Because this iterates over every single possible permutation of nodes (except 0 as the start node) in the tree until it times out, the time complexity is O((n-1)!) which simplifies down to O(n!).

Space Complexity

The space complexity is similar to the greedy algorithm in the fact that because I am using the system stack, when recursing down, my implementation only uses O(n) memory/space. This is because after I find a solution, when backtracking I pop the old solution off the stack and put a new one on. This causes my algorithm to only ever have a maximum of one full traversal stored in memory at a time, the current one it is evaluating.

Branch and Bound

```
def branch_and_bound_recursive(self,
                               currnode: int,
                               parent_rcm: list[list[float]],
                               parent_lower_bound: float,
                               visited: list[int]):

    visited.append(currnode)
    self.n_nodes_expanded += 1

    for edge_index in range(0, len(self.edges[currnode])):

        if self.edges[currnode][edge_index] == math.inf:
            continue

        if edge_index == visited[0]: # Evaluating the path to the starting
node
            if len(visited) == len(self.edges): # Complete path was found
            ...
            continue
```

```
        if edge_index in visited:
            continue

        curr_lower_bound, curr_rcm = self.calculate_reduced_cost_matrix(
            parent_rcm,
            parent_lower_bound,
            visited,
            edge_index
        )

        if curr_lower_bound >= self.BSSF:
            self.n_nodes_pruned += 1
            self.cut_tree.cut(visited)
            continue

        self.branch_and_bound_recursive(
            edge_index,
            curr_rcm,
            curr_lower_bound,
            visited.copy()
        )
```

Priority Queue

I do not have a priority queue in my standard Branch and Bound algorithm as I implemented that in the smart branch and bound algorithm. Thus the time and space complexity is O(0).

Reduced Cost Matrix, including updating it

```
def reduce_cost_matrix(self, matrix: list[list[float]]) ->
list[list[float]]:

    n = len(self.edges)
    reduction_cost = 0

    # Make sure there is a 0 in every row
    for i in range(n):
        min_value = min(matrix[i])
        if min_value != 0 and min_value != math.inf:
            min_value = min_value
            reduction_cost += min_value
            for j in range(n):
                if matrix[i][j] != math.inf:
                    matrix[i][j] -= min_value
```

```
    # Make sure there is a 0 in every column
    for j in range(n):
        min_value = min([row[j] for row in matrix])
        if min_value != 0 and min_value != math.inf:
            reduction_cost += min_value
            for i in range(n):
                if matrix[i][j] != math.inf:
                    matrix[i][j] -= min_value

    return matrix, reduction_cost
```

My reduced cost matrix is calculated at the beginning before the start of the algorithm, then when each new node is traversed, it gets passed in the parent's reduced cost matrix, which is then modified and passed to it's child if the lower bound calculated is less than the current best solution so far.

The actual updating and reducing of my cost matrix is done by the function above (reduce_cost_matrix). For each of the 2 blocks, `Make sure there is a 0 in every row` and `Make sure there is a 0 in every column`, there is a nested for loop, which in the worst case scenario will cause each block to have a $O(n^2)$. These 2 blocks added together will cause the total time and space complexity for the updating and the storage of the reduced cost matrix to be $O(n^2)$.

BSSF Initialization

I implemented the BSSF initialization using my previous greedy algorithm. Running the greedy algorithm has a time complexity of $O(n^2)$ with a space complexity of O(n).

Expanding one Search State into its children

For my implementation of branch and bound, I traverse the tree in the exact same way as my DFS implementation, except with potentially less children visited per node if the lower bound for each node calculated is greater than the BSSF. This means that for each node, my algorithm could expand up to n-1 recursive calls. The time complexity in this worst case scenario is still O(n!), but through the limiting of the lower bound, many sub problems are not being traversed down due to the reduced cost matrix making the lower bound be higher than the best solution so far. The space complexity is still O(n).

Describe the data structures you used to represent the states (i.e. how did you choose to implement the partial states, and why did you do it that way?)

My data structure that I used was very simple. For my implementation, I recursed down the tree of posibilites, and for each node where there were multiple sub nodes, those were tried one at a time until either a solution was found or until the cost of taking the sub node was deemed to be a greater than the best solution so far. This created a tree data structure using the system stack as the main storage and manager of the data layers.

The full algorithm.

For the time complexity for the full algorithm, the worst case scenario is still O(n!). The traveling salesman problem is still NP, a problem that is solvable in polynomial time only by a nondeterministic Turing Machine, not a deterministic one. This means that any solution on a real computer will not be able to solve it in polynomial time.

However, by significantly shrinking the possibilities that can be taken due to a much faster $O(n^2)$ lower bound calculation algorithm compared to subsequent O(n!) calculation algorithm that would occur otherwise. This still has a worst case scenario calculation of O(n!), but a real life answer of much faster than O(n!) although that is hard to generalize due to the various real world factors such as initial graph density and spread of weights.

The space complexity is still O(n).

Smart Branch and Bound

```
def smart_branch_and_bound_recursive(self,
                             currnode: int,
                             parent_rcm: list[list[float]],
                             parent_lower_bound: float,
                             visited: list[int]):

    visited.append(currnode)

    edges_to_visit = self.edges[currnode]
    sorted_edges = sorted(
        range(
            len(edges_to_visit)),
            key=lambda i: edges_to_visit[i]
        )
    pruned_sorted_edges = []

    for sorted_edge_index in range(0, len(sorted_edges_to_visit) - 1):
        if (
            self.edges[currnode][sorted_edges[sorted_edge_index]] ==
math.inf or
            self.edges[currnode][sorted_edges[sorted_edge_index]] == 0
```

```
            ):
                continue
            pruned_sorted_edges.append(sorted_edges[sorted_edge_index])

        for edge_index in pruned_sorted_edges_to_visit:

            if self.edges[currnode][edge_index] == math.inf:
                continue

            if edge_index == visited[0]: # Evaluating the path to the starting
node
                if len(visited) == len(self.edges): # Complete path was found
                ...
                continue

            if edge_index in visited:
                continue

            curr_lower_bound, curr_rcm = self.calculate_reduced_cost_matrix(
                parent_rcm,
                parent_lower_bound,
                visited,
                edge_index
            )

            if curr_lower_bound >= self.BSSF:
                self.n_nodes_pruned += 1
                self.cut_tree.cut(visited)
                continue

            self.smart_branch_and_bound_recursive(
                edge_index,
                curr_rcm,
                curr_lower_bound,
                visited.copy()
            )
```

Standard Branch and Bound vs Smart Branch and Bound

My smart branch and bound is almost exactly identical to my standard branch and bound algorithm, the only difference is the order of the edges traversed. My standard branch and bound algorithm uses the edges as they are stored, in an arbitrary ascending order starting at edge 0 and going to edge n. My smart branch and bound algorithm takes the initial list of edges and sorts them from the least cost to the greatest cost as well as pre-pruning the 0 values and infinities from the list that the main algorithm iterates over.

This does not effect the overall worst case time or space complexity, but unlike the standard branch and bound algorithm, the smart branch and bound algorithm has much higher chance of encountering better paths with lower costs given the same time as the standard branch and bound algorithm. Even though the time and space complexity is identical, given a certain time frame, the smart branch and bound algorithm should find a lower cost solution faster than the standard branch and bound algorithm.

Priority Queue Data Structure

```
# From the function above
    edges_to_visit = self.edges[currnode]
    sorted_edges = sorted(
        range(
            len(edges_to_visit)),
            key=lambda i: edges_to_visit[i]
        )
    pruned_sorted_edges = []

    for sorted_edge_index in range(0, len(sorted_edges_to_visit) - 1):
        if (
            self.edges[currnode][sorted_edges[sorted_edge_index]] ==
math.inf or
            self.edges[currnode][sorted_edges[sorted_edge_index]] == 0
            ):
            continue
        pruned_sorted_edges.append(sorted_edges[sorted_edge_index])
```

For my implementation, I did not use a standard priority queue data structure, but instead I used the stack to handle each layer, and sorted from least cost to greatest cost. I then traversed down the layers in a depth first - style search, taking the least cost path each time, the backtracking when hitting a dead end. This allows my smart branch and bound algorithm to be most efficient with the paths it travels given the time, while also digging deep and coming up with solutions quickly as opposed to a breath first search style approach.

Priority Computation

As listed above, my priority is determined by cost of each outgoing edge. This is sorted from least to greatest, allowing the algorithm to first traverse down more promising paths before traversing less promising ones. This allows the more promising paths to be traversed first, potentially finding a better solution given a certain amount of time compared to just using the provided order like the DFS and the standard branch and bound algorithms use.

With this priority system, I was hoping to be able to properly balance finding the best path, trying to find and return solutions quickly and using a low complexity priority sorting system, O(n log(n)). I think that this strategy does work the way that I hoped it would. It potentially could be better optimized for drilling down deep quickly (if a really short time limit is needed) or finding a better solution as a first solution, but overall because of the simplicity of the algorithm I am happy with how it is performing.

The space complexity overall is still O(n).

Empirical Results

Runtimes

Seed	N	Random	Greedy	DFS	B&B	Smart B&B
312	10	3.376	3.411	3.376	3.376	3.376
1	15	5.086	4.647	5.419	3.98	3.98
2	20	6.834	4.265	8.203	3.866	3.866
3	30	12.091	6.121	11.421	6.056	5.386
4	50	24.747	8.095	22.53	8.095	7.245
64	22	8.028	5.676	7.35	4.455	4.455
64	24	9.392	5.358	9.259	5.327	5.281
64	26	10.225	5.642	11.206	5.642	5.262
64	200	N/A	15.325	95.903	15.325	15.325

At what point does your DFS algorithm fail to find a solution that is better than Greedy, but the other algorithms still do?

- It appears that the point at which this happens is between N=10 and N=15. At N=10, the DFS algorithm was still able to traverse every single possibility within the 60 seconds. At N=15, it was cut short, thus getting the worst score of all the possible algorithms.

At what point does your B&B algorithm fail to improve upon your initial BSSF?

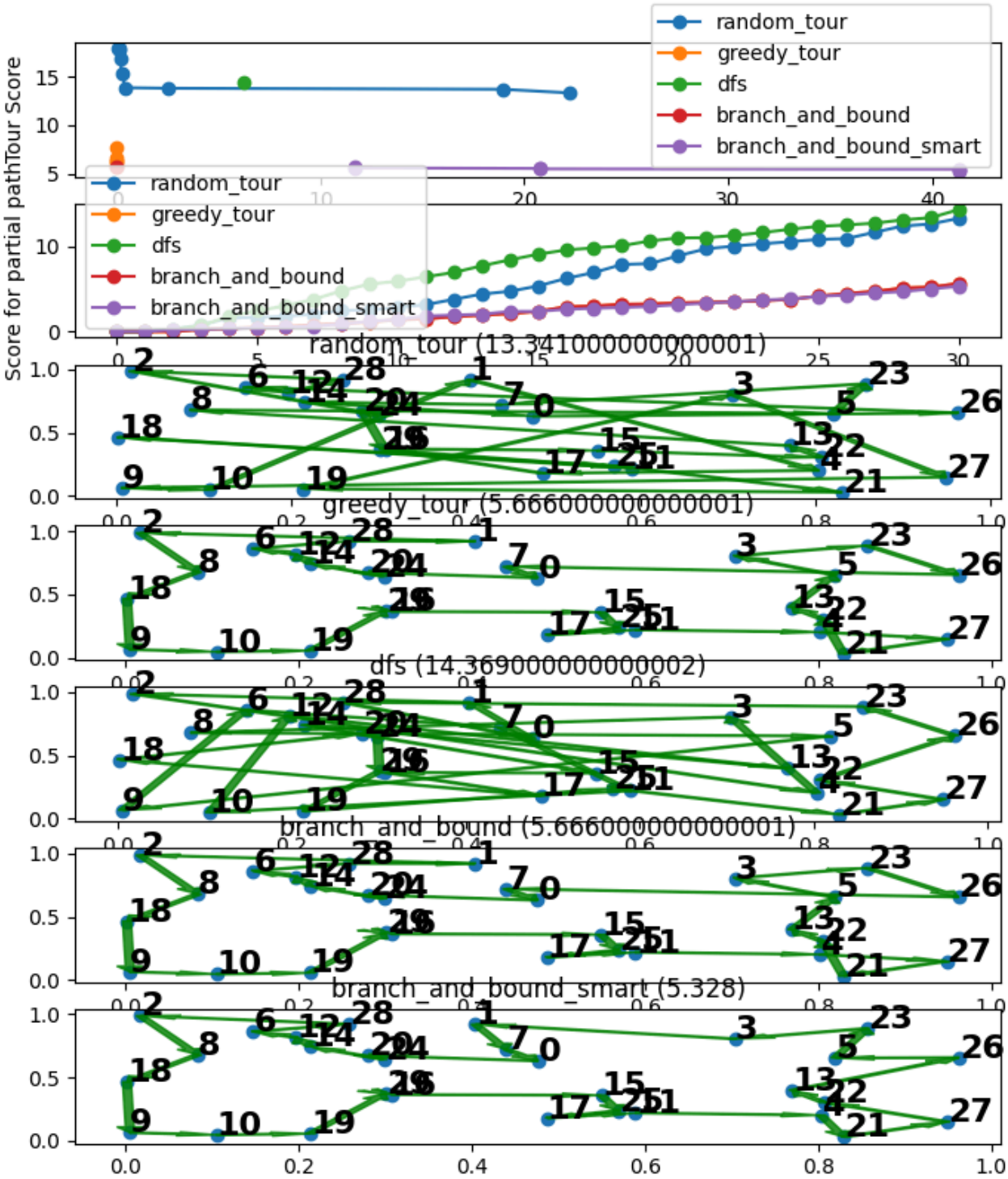
- It appears from my empirical data that the point at which branch and bound fails to improve on the initial BSSF set by the greedy algorithm at the 60 second cutoff is between N=24 and N=26

Is there a point at which your smart B&B algorithm fails to provide a better solution than your simple B&B implementation?

- From my testing, I did not find any time in which my standard branch and bound algorithm outperformed my smart branch and bound algorithm. At N=200 was the only point in which they returned the same score, and that was because they both returned the initial BSSF calculated by the greedy algorithm because neither of them was able to find a better solution in 60 seconds.

Discussion

This is the result for running with N=30



I found N~30 to be around the point in which the normal branch and bound failed to improved on the greedy algorithm and the smart branch and bound still was able to improve.

Strengths and Weaknesses

I would say that the results speak for themselves, if you have a small set of nodes and you want the exact answer, then use DFS for the simplicity (or use smart branch and bound if it

available). For a large set of nodes, if you want a quick answer then chose the greedy algorithm. If you want the best answer given a certain amount of time, then use the smart branch and bound algorithm.

From my testing and experience, there does not appear to be a reason to use the random algorithm or the normal branch and bound if you have access to all the different algorithms, although they are simpler to implement than their counter parts if you value the time for implementation over the time for execution.