Project 5 Report

Complexity Analysis

Greedy

```python
def greedy_tour(edges: list[list[float]], timer: Timer) -> list[SolutionStats]:
    nodes_expanded = 0
    nodes_pruned = 0
    cut_tree = CutTree(len(edges))
    for start in range(len(edges)):
        nodes_expanded += 1
        temp_tour = [start]
        temp_score = 0
        current = start
        solution_found = False
        rows_left = list(range(len(edges)))
        while True:
            min_edge = inf
            best_dest = None
            rows_left.remove(current)
            for dest in rows_left:
                if edges[current][dest] < min_edge:
                    min_edge = edges[current][dest]
                    best_dest = dest
            if best_dest is None:
                cut_tree.cut(temp_tour)
                nodes_pruned += 1
                break
            temp_tour.append(best_dest)
            temp_score += min_edge
            current = best_dest
            if is_solution(temp_tour, edges):
                solution_found = True
                temp_score += edges[current][start]
                break
        if solution_found:
            return [...]
    return [...]
```

The greedy algorithm has a simple approach: start at a node, pick the cheapest edge, and repeat. If a solution isn't found, do the same thing starting at a different starting node until a solution is found or until you run out of starting nodes. The "for dest in rows_left:" loop will have, on average, a time complexity of O(n). The "while True:" loop will repeat, at most, n times and will thus have a time complexity of O(n^2). The "for start in range(len(edges)):" loop will repeat, at most, n times and will thus have a time complexity of O(n^3). This algorithm will only have to store an additional list of "row_left" and "temp_tour" which will have at most n elements. Therefore, the greedy algorithm will have a space complexity of O(n) and time complexity of O(n^3). The greedy algorithm, as well as all the other algorithms, use the "is_solution" function which calculates whether a tour is the correct length and can return the beginning. It has a space and time complexity of O(1).

```
def is_solution(tour: list[float], edges: list[list[float]]):
    if len(tour) != len(edges):
        return False
    if math.isinf(edges[tour[-1]][tour[0]]):
        return False
    return True
```

DFS

The DFS algorithm performs a Depth-first search starting at 0 and finds the best tour. For every path from the current node, it puts that many nodes on the stack to be explored. Because of this, the resulting stack / tree will have n! nodes on it. This extreme inefficiency outweighs all other time and space costs in the algorithm. Therefore the DFS algorithm has a time and space complexity of O(n!)

```python
def dfs(edges: list[list[float]], timer: Timer) -> list[SolutionStats]:  4 usages
    solution_stats = []
    max_queue_size = 1
    nodes_expanded = 0
    cut_tree = CutTree(len(edges))
    best_score = math.inf
    stack = [([0], 0)]
    while len(stack) > 0:
        tour, score = stack.pop(0)
        nodes_expanded += 1
        current = tour[-1]
        for dest in range(len(edges)):
            if dest in tour:
                pass
            else:
                new_tour = tour + [dest]
                new_score = score + edges[current][dest]
                if is_solution(new_tour, edges) and new_score < best_score:
                    best_score = new_score + edges[dest][0]
                    solution_stats.append(SolutionStats(
                        tour=new_tour,
                        score=best_score,
                        time=timer.time(),
                        max_queue_size=max_queue_size,
                        n_nodes_expanded=nodes_expanded,
                        n_nodes_pruned=0,
                        n_leaves_covered=cut_tree.n_leaves_cut(),
                        fraction_leaves_covered=cut_tree.fraction_leaves_covered()
                    ))
                else:
                    stack.insert( __index: 0,  __object: (new_tour, new_score))
                    max_queue_size = max(max_queue_size, len(stack))
    if len(solution_stats) == 0:...
    return solution_stats
```

Disclaimer

For my B&B and B&B Smart algorithms. I had originally used a stack on B&B and a priority queue on B&B Smart. However, the overhead of using a priority queue (mainly the O(log(n)) inserts) caused the B&B Smart algorithm to be slower than the B&B algorithm, despite still

being faster than Professor Bean's B&B Smart algorithm. I spoke to Professor Bean and he told me to switch the algorithms. So if you're confused on why my B&B uses a priority queue and my B&B Smart uses a stack, that's why.

Reduced Cost Matrix

```
def create_rcm(rcm: dict[tuple[int, int], float], initial_lb: float,  3 usages
               rows_left: list[int], cols_left: list[int]) -> tuple[dict[tuple[int, int], float], fl
    for row in rows_left:
        min_row_edge = math.inf
        for col in cols_left:
            min_row_edge = min(min_row_edge, rcm[(row, col)])
        for col in cols_left:
            rcm[(row, col)] = rcm[(row, col)] - min_row_edge
        initial_lb += min_row_edge
    for col in cols_left:
        min_col_edge = math.inf
        for row in rows_left:
            min_col_edge = min(min_col_edge, rcm[(row, col)])
        for row in rows_left:
            rcm[(row, col)] = rcm[(row, col)] - min_col_edge
        initial_lb += min_col_edge
    return rcm, initial_lb
```

```
def update_rcm(initial_rcm: dict[tuple[int, int], float], initial_lb: float,  2 usages
               rows_left: list[int], cols_left: list[int], current: int, dest: int) ->
    initial_lb += initial_rcm[(current, dest)]
    if initial_lb == math.inf:
        return initial_rcm, initial_lb
    else:
        rcm = initial_rcm.copy()
        for row in rows_left:
            rcm.pop((row, dest))
        for col in cols_left:
            rcm.pop((current, col))
        rcm.pop((current, dest))
        rcm[(dest, current)] = math.inf
        return create_rcm(rcm, initial_lb, rows_left, cols_left)
```

There are two functions that both B&B and B&B Smart used to calculate and update the reduced cost matrices. The "create_rcm" function goes through the rows remaining on the RCM to find

the minimum edge on each row and then does the same for each column, creating a new RCM and computing a new lower bound. If “m” represents the number of rows/columns left in the RCM then this function has a time complexity of O(m^2) and a space complexity of O(1). The “update_rcm” function simulates taking an edge from “current” to “dest”. By doing so it makes a copy of the RCM, removes the “current” row and the “dest” column, and calculates a new RCM and lower bound by calling “create_rcm”. Copying the RCM takes (m^2) time, removing the column and row takes (m) time, and “create_rcm” takes (m^2) time. Therefore, “update_rcm” has a time and space complexity of O(m^2). The reason I use “m” is because it relates better to “n” from the functions that call it. These functions would be very inefficient if “m” was always equal to “n”. However, as the B&B and B&B Smart algorithms get deeper into the search, “m” decreases. Specifically, “m” will always be “n” minus “len(tour)”.

B&B

The Branch and Bound algorithm uses a mixture of a stack and a priority queue to keep track of which states to explore. This algorithm also introduces one of the most important concepts that sets it apart from DFS: pruning. By calling “udpate_rcm” on every child of a state, it is able to tell which children are more likely to have a better solution. The algorithm prunes every child that has a lower bound higher than the “bssf” and otherwise saves it to explore later.

Priority Queue - This algorithm uses a heapq to store all the promising children of the current node being explored. The heapq prioritizes the children by the highest lower bound. When a node is finished being explored the heapq inserts all the children in that order onto the stack. That way, the first node to be popped off the stack will always be at the deepest level, and will have the lowest lower bound of that level. The heapq has inserts and pops of O(log(n)). By limiting the heapq to only the children of the node being explored, it guarantees that “n” in this

case is never greater than the number of destinations in the problem. The stack has inserts and pops of O(1) which is good because the stack will be much bigger than the number of destinations.

BSSF - The bssf is initialized by using the greedy algorithm. The greedy algorithm has a time complexity of O(n^3) which is much faster than the time complexity of B&B.

Expanding Search States - When the algorithm explores a node that has been popped off the stack, it has a lower bound and an rcm already. For each child, the algorithm creates a new lower bound and rcm and updates the number of rows and columns left. This information is all stored on the stack when the child is pushed onto the stack. If “m” is the number of rows/columns left, the time and space complexity of expanding a search state is O(m^3)

State Representation - The state is represented by a tuple that contains the following: a priority score (not used outside of heapq functionality), an updated reduced cost matrix, a lower bound, a list of rows left in the rcm, a list of columns left in the rcm, and the route taken so far. Doing it this way allows me to not have to create a new data structure to store these elements.

With all that being said, calculating the overall time and space complexity can be a little difficult because it depends on how many of the states can be pruned by the algorithm. If “b” represents the amount of states pruned then this can turn the problem of exploring (n!) solutions into a problem of exploring (b^n) solutions. Additionally, this algorithm receives some benefit for keeping track of the number of rows/columns left, which has previously been denoted as “m”. The great thing about this is that the number of leaves grows exponentially as we go deeper down the tree of solutions, which means that the amount of states that have a large “m” is small compared to the number of states that have a small “m”. I would say the total complexity this algorithm is O((b^n) * (m^3)).

```python
def branch_and_bound(edges: list[list[float]], timer: Timer) -> list[SolutionStats]:  5 usages
    n = len(edges)
    solution_stats = []
    cut_tree = CutTree(n)
    nodes_pruned, nodes_expanded, max_queue_size = 0, 0, 1
    bssf = greedy_tour(edges, timer)[0].score
    rows_left = list(range(n))
    cols_left = list(range(n))
    initial_rcm = dict()
    for row in rows_left:
        for col in cols_left:
            initial_rcm[(row, col)] = edges[row][col]
    initial_rcm, initial_lb = create_rcm(initial_rcm, initial_lb: 0, rows_left, cols_left)
    stack = [(0, initial_rcm, initial_lb, rows_left, cols_left, [0])]
    while len(stack) > 0:
        if timer.time_out():
            return solution_stats
        _, rcm, lb, rows_left, cols_left, tour = stack.pop()
        if lb > bssf:
            nodes_pruned += 1
            cut_tree.cut(tour)
        else:
            nodes_expanded += 1
            current = tour[-1]
            children = []
            for dest in [x for x in cols_left if x != 0]:
                new_rows_left = [x for x in rows_left if x != current]
                new_cols_left = [x for x in cols_left if x != dest]
                new_rcm, new_lb = update_rcm(initial_rcm, initial_lb,
                                             new_rows_left, new_cols_left, current, dest)
                if new_lb > bssf:
                    nodes_pruned += 1
                    cut_tree.cut(tour)
                else:
                    new_tour = tour + [dest]
                    if is_solution(new_tour, edges):
                        bssf = new_lb
                        solution_stats.append(SolutionStats(
                            tour=new_tour,
                            score=new_lb,
                            time=timer.time(),
                            max_queue_size=max_queue_size,
                            n_nodes_expanded=nodes_expanded,
                            n_nodes_pruned=nodes_pruned,
                            n_leaves_covered=cut_tree.n_leaves_cut(),
                            fraction_leaves_covered=cut_tree.fraction_leaves_covered()
                        ))
                    else:
                        heapq.heappush( *args: children, ((random.random()/100) - new_lb,
                                                          new_rcm, new_lb, new_rows_left, new_cols_left, new_tour))
            while len(children) > 0:
                stack.append(heapq.heappop(children))
            max_queue_size = max(max_queue_size, len(stack))
    if len(solution_stats) == 0:...
    return solution_stats
```

B&B Smart

```
def branch_and_bound_smart(edges: list[list[float]], timer: Timer) -> list[SolutionStats]:  8 usage
    n = len(edges)
    solution_stats = []
    cut_tree = CutTree(n)
    nodes_pruned, nodes_expanded, max_queue_size = 0, 0, 1
    bssf = greedy_tour(edges, timer)[0].score
    rows_left = list(range(n))
    cols_left = list(range(n))
    initial_rcm = dict()
    for row in rows_left:
        for col in cols_left:
            initial_rcm[(row, col)] = edges[row][col]
    initial_rcm, initial_lb = create_rcm(initial_rcm,  initial_lb:  0, rows_left, cols_left)
    stack = [(initial_rcm, initial_lb, rows_left, cols_left, [0])]
    while len(stack) > 0:
        if timer.time_out():
            return solution_stats
        rcm, lb, rows_left, cols_left, tour = stack.pop()
        if lb > bssf:
            nodes_pruned += 1
            cut_tree.cut(tour)
        else:
            nodes_expanded += 1
            current = tour[-1]
            for dest in [x for x in cols_left if x != 0]:
                new_rows_left = [x for x in rows_left if x != current]
                new_cols_left = [x for x in cols_left if x != dest]
                new_rcm, new_lb = update_rcm(initial_rcm, initial_lb,
                                             new_rows_left, new_cols_left, current, dest)
                if new_lb > bssf:
                    nodes_pruned += 1
                    cut_tree.cut(tour)
                else:
                    new_tour = tour + [dest]
                    if is_solution(new_tour, edges):
                        bssf = new_lb
                        solution_stats.append(SolutionStats(
                            tour=new_tour,
                            score=new_lb,
                            time=timer.time(),
                            max_queue_size=max_queue_size,
                            n_nodes_expanded=nodes_expanded,
                            n_nodes_pruned=nodes_pruned,
                            n_leaves_covered=cut_tree.n_leaves_cut(),
                            fraction_leaves_covered=cut_tree.fraction_leaves_covered()
                        ))
                    else:
                        stack.append((new_rcm, new_lb, new_rows_left, new_cols_left, new_tour))
                        max_queue_size = max(max_queue_size, len(stack))
    if len(solution_stats) == 0:...
    return solution_stats
```

The Branch and Bound Smart algorithm is only different from the B&B algorithm in its implementation of the stack. While the B&B algorithm uses a combination of a stack and a priority queue, the B&B Smart algorithm only uses a stack. This is because of the overhead that comes from using a priority queue. While the priority queue does help explore the most promising child of the node, it adds some complexity because of the O(log(n)) inserts and pops. While log(n) isn't that bad, when you have to explore (b^n) different solutions, that small difference can add up. So, the overhead of using a priority queue, in my experience, was not worth the advantages that it could bring. Instead, much like the DFS algorithm, the B&B Smart algorithm uses a single stack. Every time a viable child is found, it inserts it at the end of the stack, guaranteeing that deeper states are always explored before shallow states. Much like the B&B algorithm, I would say that the overall complexity would also be O((b^n) * (m^3)). However the difference comes from some small adjustments that remove a lot of overhead which is accounted for in some constants or other terms in the complexity of each algorithm.

Empirical Results

Seed	N	Random	Greedy	DFS	B&B	Smart B&B
312	10	3.376	3.783	4.005	1.16	1.253
1	15	5.086	5.131	na	1.298	1.125
2	20	6.945	4.419	9.326	1.421	1.336
3	30	12.091	7.015	na	1.989	1.541
4	50	24.747	8.715	na	2.334	2.192
122	75	na	10.455	na	3.377	2.938

122	100	na	12.604	na	2.673	2.558
122	150	na	13.191	na	na	na

The DFS fails to find a solution after n=20. The B&B only algorithm failed to improve on the initial BSSF on the final test where n=150. My B&B Smart algorithm is better than my B&B in all tests except the first one where n=10. This data shows some important things about these algorithms. DFS is obviously the least efficient and will timeout before finding an answer once n grows even slightly large. The greedy algorithm appears to always be able to find a solution, even if the solution is nowhere near ideal, even when B&B and B&B Smart can't find a better solution. I also noticed that the B&B Smart typically starts out with worse solutions but eventually surpasses the B&B solutions, as seen below: