Project 5 Report

[20] Include your well-commented code.
See the Code Snippets section below.

[10] Explain both the time and space complexity of your algorithm by showing and summing up the complexity of each subsection of your code. Keep in mind the following things:

- Priority Queue
- SearchStates
- Reduced Cost Matrix, and updating it
- BSSF Initialization
- Expanding one SearchState into others
- The full Branch and Bound algorithm

NOTE: In my explanations, I use the term "distance table" to refer to Reduced Cost Matrices. This is so that my explanations match my code more exactly. This is different from DistanceTable objects, which are my SearchStates.

Greedy Algorithm for Finding Initial Upper Bound and BSSF
Time: O(n^3), Space: O(n^2)
The O(n^3) time complexity of my greedy algorithm comes from running my solve_greedy() function n times, using each city as my start city. The solve_greedy() function runs in O(n^2) time because it finds the nearest city (out of n cities) for each of the n cities, an O(n^2) time operation.
The O(n^2) space complexity of my greedy algorithm comes primarily from creating deep copies of each of my states (DistanceTable objects), which each have O(n^2) integers stored in their distance tables.

Reduction Algorithm, used to find initial lower bound
Time: O(n^2), Space: O(1)
The O(n^2) time complexity of my reduction algorithm is a result of iterating through each of the n items in n rows to reduce them. I did the same to reduce the columns.
The O(1) space complexity comes from not allocating any additional data. Data is only modified by the reduction algorithm.

Branch and Bound Algorithm
*Time: O(n^2 * b^n), Space: O(n^2 * b^n)*
For my algorithm's time complexity, the n^2 part represents initializing n^2 elements in my distance table b^n times. Pessimistically, there would be n! states, but my algorithm prunes enough to reduce the total number of states. The variable b stands for "branching factor", which equals (total nodes created - total nodes pruned) / total nodes expanded.
For my algorithm's space complexity, the n^2 part represents the n^2 elements in my distance table, which is initialized b^n times. My empirical data shows that my space complexity is

*actually less than O(n^2 * b^n), but I'm using a high estimate with the acknowledgement that as many as b^n can potentially be pushed to my priority queue.*

Expansion
Time: O(n^3), Space: O(n^3)
The time complexity of the expansion portion of my branch and bound algorithm comes from making O(n) deep copies of the parent distance table, each with n^2 elements. It also comes from reducing each of those n copies, a O(n^2) operation (see above).
The space complexity of this portion of my algorithm primarily comes from creating n^2 elements in n deep copies of the parent distance table.

[5] Describe the data structures you use to represent the states.
To represent the states of my algorithm, I created the DistanceTable class. The class has a two-dimensional array of integers that stores the current distances for that state. It also has other attributes such as a list of indexes for the route, a list of the indexes of unvisited cities, and that state's current lower bound (cost).
Instead of each state containing a list of city objects, the DistanceTable class only contains an integer for the number of cities. City objects are only referenced to initialize the first DistanceTable and to produce a TSPSolution object. This change improved my algorithm's performance by a factor of ~2.

[5] Describe the priority queue data structure you use and how it works.
As a priority queue, I pushed my DistanceTable objects to a heapq. I then overrode the < operator by defining __lt__() in my DistanceTable class to return $\frac{self.lowerbound}{self.depth} < \frac{other.lowerbound}{other.depth}$*, where 'other' is some other DistanceTable object that the current DistanceTable is being compared to. My __lt__() function gives priority to DistanceTable objects with a low lower bound and a high depth, since a heapq is sorted in ascending order. This prioritization helps promote the cheapest states that are also close to a leaf node, which results in more updates to the upper bound and earlier pruning.*

[5] Describe your approach for the initial BSSF.
I use a greedy algorithm to determine my initial BSSF. The algorithm begins from the first city and repeatedly traverses the cheapest edge until it creates a cycle, if one exists. The algorithm then repeats this process, using each city as the start city. Eventually a reasonably cheap cycle is found. See the time and space complexity descriptions above for more details.

[25] Include a table containing the following columns.

# 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	1.87	10534*	45	7	11687	10115
16	902	5.74	7954*	80	7	32046	28292
14	731	21.59	7432*	62	6	148927	124011
17	710	37.33	9758*	94	17	189831	165261
40	280	60.00	15811	808	7	64351	55476
34	873	60.00	14299	462	15	87669	76683
16	697	54.38	9354*	93	5	301950	262432
28	437	60.00	14513	606	4	124429	106418
46	749	60.00	19852	821	9	48891	42307
13	299	1.38	10122*	56	2	11032	9405

The first two rows of the table used Hard (Deterministic) mode. The rest of the table used Hard mode with random city counts and seeds. This is so the first two lines are comparable with other projects while the rest simply shows functionality and trends.

[10] Discuss the results in the table and why you think the numbers are what they are, including how time complexity and pruned states vary with problem size.

Total Number of States

Although I expected larger problems to generate more states, that wasn't always true. My largest problems with 34, 40, and 46 cities generated less than 100,000 states each. This is because generating states with distance tables that are larger takes larger portions of the allotted 60 seconds. Therefore, especially small and especially large problem sizes resulted in less states. The most states (301950) were generated by a problem size of 16 with seed 697.

Total Number of Pruned States

The total number of pruned states followed the total number of states closely and consistently, being an average of 86.31% of the size. This high proportion makes sense since the algorithm is built to prune aggressively for how it updates its upper bound early and often.

Time Complexity

The larger problems had the highest time complexity, hitting the 60 second time allotment much more frequently. The empirical results support my time complexity analysis of my branch and bound algorithm (above).

[10] Discuss the mechanisms you tried and how effective they were in getting the state space search to dig deeper and find more solutions early.

Initially, I sorted my priority queue by the lower bound of each state. However, solving for a problem size of 14 with seed 2 took over 60 seconds to complete. This was a disappointingly ineffective way of prioritizing states.
To improve my performance, I sorted my priority queue by the quotient between the lower bound and the depth of each state (lower bound/depth). This change promoted states that are deeper in the expansion tree so that I evaluate leaf nodes and find solutions earlier. The same problem with 14 cities in seed 2 took 5.25 seconds, an incredible performance improvement.

Code Snippets

An excerpt from my TSPSolver class

```
# ------------------------------------------------------------------------ #
#                                                                          #
#                                Optimization                              #
#                                                                          #
# ------------------------------------------------------------------------ #
# Included for optimization. This type of function would typically be found in the
DistanceTable constructor.
# It's removal means that I only have to call len(cities) once per algorithm. It
also means that an integer is
# stored where a pointer would have been stored in the DistanceTable class. The
change resulted in a surprising 50%
# speedup.
def initialize_dist_table(self) -> DistanceTable:
    cities = self._scenario.get_cities()
    n_cities = len(cities)
    distances = []

    for i in range(n_cities):
        distances.append([])

        for j in range(n_cities):
            distances[i].append(cities[i].cost_to(cities[j]))

    return DistanceTable(n_cities, distances)

# Also included for optimization, as described above.
# Time: O(n), Space: O(n)
def dist_table_to_solution(self, dist_table: DistanceTable) -> TSPSolution:
    cities = self._scenario.get_cities()
    route_cities = []

    # Convert route indexes to cities
    for index in dist_table.route:
        route_cities.append(cities[index])
```

```
    return TSPSolution(route_cities)

# ------------------------------------------------------------------------ #
#                                                                          #
#                                  Greedy                                  #
#                                                                          #
# ------------------------------------------------------------------------ #
# A helper function for the greedy algorithm
# Time: O(n^2), Space: O(n)
def solve_greedy(self, start_city_index: int, dist_table: DistanceTable) ->
TSPSolution:
    dist_table.set_start_city(start_city_index)
    next_city_index = dist_table.get_nearest_city() # Time: O(n), Space: O(1)
    solution = None

    # Time: O(n^2), Space: O(1)
    while next_city_index is not None:
        dist_table.visit(next_city_index)
        next_city_index = dist_table.get_nearest_city()

    # Time: O(n), Space: O(n)
    if dist_table.has_solution():
        solution = self.dist_table_to_solution(dist_table)

    return solution

# Time: O(n^3), Space: O(n^2)
def greedy(self, time_allowance=60.0):
    dist_table = self.initialize_dist_table()
    solution_count = 0
    bssf = None
    results = {}

    start_time = time.time()

    # Time: O(n^3), Space: O(n^2)
    for city_index in range(dist_table.n_cities):       # Iterates O(n) times
        if time.time() - start_time > time_allowance:
            break

        # Time: O(n^2), Space: O(n^2)
        # "Deep copies" dist_table of O(n^2) size
        current_solution = self.solve_greedy(city_index, copy.deepcopy(dist_table))

        if current_solution is not None:
            solution_count += 1
```

```
            if bssf is None or current_solution.cost < bssf.cost:
                bssf = current_solution

    end_time = time.time()

    results['cost'] = bssf.cost if solution_count > 0 else math.inf
    results['time'] = end_time - start_time
    results['count'] = solution_count
    results['soln'] = bssf
    results['max'] = None
    results['total'] = None
    results['pruned'] = None

    return results

# ------------------------------------------------------------------------------ #
#                                                                                #
#                                Branch and Bound                                #
#                                                                                #
# ------------------------------------------------------------------------------ #
# Time: O(n), Space: O(n)
def prune(self, priority_queue, upper_bound):
    pruned_queue = []

    for i in range(len(priority_queue)):
        if priority_queue[i].lower_bound < upper_bound:
            pruned_queue.append(priority_queue[i])

    return pruned_queue, len(priority_queue) - len(pruned_queue)

def branch_and_bound(self, time_allowance=60.0):
    dist_table = self.initialize_dist_table()
    priority_queue = []
    bssf = None
    solution_count = 0
    max_queue_size = 0
    pruned_count = 0
    total_states = 0
    results = {}

    # Find initial upper bound and solution
    greedy_results = self.greedy(time_allowance)
    upper_bound = greedy_results['cost']
    bssf = greedy_results['soln']

    start_time = time.time()
```

```
# Find initial lower bound
dist_table.set_start_city(0)
dist_table.reduce()
heapq.heappush(priority_queue, dist_table)

# Solve
# Time: O(n^2 * b^n), Space: O(n^2 * b^n)
while len(priority_queue) > 0 and time.time() - start_time < time_allowance:
    parent_table = heapq.heappop(priority_queue)

    # Time : O(n^3), Space: O(n^3)
    for city_index in parent_table.unvisited:
        # Time: O(n^2), Space: O(n^2)
        branch_table = copy.deepcopy(parent_table)
        total_states += 1

        # Time: O(n), Space: O(1)
        branch_table.visit(city_index)

        # Time: O(n^2), Space: O(1)
        branch_table.reduce()

        # print(branch_table)

        # Evaluates solution
        if branch_table.has_solution():
            branch_table.finalize_lower_bound()

            # Saves solution, else prunes
            if branch_table.lower_bound < upper_bound:
                upper_bound = branch_table.lower_bound
                bssf = self.dist_table_to_solution(branch_table)
                solution_count += 1
                self.prune(priority_queue, upper_bound)
            else:
                pruned_count += 1

        # Evaluates branch
        elif branch_table.lower_bound < upper_bound:
            heapq.heappush(priority_queue, branch_table)
            max_queue_size = max(max_queue_size, len(priority_queue))

        # Prunes branch
        else:
            pruned_count += 1
```

```
    end_time = time.time()

    results['cost'] = bssf.cost if solution_count > 0 else math.inf
    results['time'] = end_time - start_time
    results['count'] = solution_count
    results['soln'] = bssf
    results['max'] = max_queue_size
    results['total'] = total_states
    results['pruned'] = pruned_count

    return results
```

My DistanceTable class, which I used as states

```
from typing import Optional, Any

import math


class DistanceTable:
    def __init__(self, n_cities: int = 0, distances: list = None):
        self.n_cities = n_cities
        self.distances = distances
        self.route = []
        self.unvisited = [*range(n_cities)]
        self.lower_bound = 0

    # Time: O(n), Space: O(1)
    def set_start_city(self, start_city_index: int) -> None:
        self.unvisited.remove(start_city_index)
        self.route.append(start_city_index)

    # ------------------------------------------------------------------------ #
    #                                                                          #
    #                                Visit a city                              #
    #                                                                          #
    # ------------------------------------------------------------------------ #
    # Time: O(n), Space: O(1)
    def visit(self, to_index: int) -> None:
        from_index = self.route[-1]
        distance = self.distances[from_index][to_index]

        # Updates distance table
        self.infinity_row(from_index)
        self.infinity_column(to_index)
        self.infinity_inverse(from_index, to_index)

        # Transfers unvisited to route
        self.unvisited.remove(to_index)
        self.route.append(to_index)
```

```
        # Updates lower bound
        self.lower_bound += distance

    # Time: O(n), Space: O(1)
    def infinity_row(self, row: int) -> None:
        row = self.distances[row]

        for i in range(len(row)):
            if row[i] != math.inf:
                row[i] = math.inf

    # Time: O(n), Space: O(1)
    def infinity_column(self, column: int) -> None:
        for row in self.distances:
            if row[column] != math.inf:
                row[column] = math.inf

    # Time: O(1), Space: O(1)
    def infinity_inverse(self, from_index: int, to_index: int) -> None:
        self.distances[to_index][from_index] = math.inf

    # ------------------------------------------------------------------------ #
    #                                                                          #
    #                                Reduce Table                              #
    #                                                                          #
    # ------------------------------------------------------------------------ #
    # Time: O(n^2), Space: O(1)
    def reduce(self) -> None:
        self.reduce_rows()
        self.reduce_columns()

    # Time: O(n^2), Space: O(1)
    def reduce_rows(self) -> None:
        for row in self.distances:
            min_distance = math.inf
            found_zero = False

            # Find the minimum distance in the row
            for distance in row:
                if distance == 0:
                    found_zero = True
                    break
                elif distance < min_distance:
                    min_distance = distance

            # Skips row if a zero was found for efficiency
            if found_zero:
                continue

            # Subtract the minimum distance from each distance in the row
            if min_distance < math.inf:
                for i in range(len(row)):
```

```
                row[i] -= min_distance

            self.lower_bound += min_distance

    # Time: O(n^2), Space: O(1)
    def reduce_columns(self) -> None:
        for j in range(self.n_cities):
            min_distance = math.inf
            found_zero = False

            # Find the minimum distance in the column
            for i in range(self.n_cities):
                distance = self.distances[i][j]

                if distance == 0:
                    found_zero = True
                    break
                elif distance < min_distance:
                    min_distance = distance

            # Skips column if a zero was found for efficiency
            if found_zero:
                continue

            # Subtract the minimum distance from each distance in the column
            if min_distance < math.inf:
                for i in range(len(self.distances)):
                    self.distances[i][j] -= min_distance

                self.lower_bound += min_distance

    # ------------------------------------------------------------------------ #
    #                                                                          #
    #                                  Greedy                                  #
    #                                                                          #
    # ------------------------------------------------------------------------ #
    # Time: O(n), Space: O(1)
    def get_nearest_city(self) -> Optional[int]:
        nearest_city_index = None

        if len(self.route) < self.n_cities:
            row = self.distances[self.route[-1]]
            min_distance = math.inf

            for i in range(len(row)):
                if i != self.route[0] and row[i] < min_distance:
                    nearest_city_index = i
                    min_distance = row[i]

        return nearest_city_index

    # ------------------------------------------------------------------------ #
```

```
    #                                                                           #
    #                                  Solution                                 #
    #                                                                           #
    # ------------------------------------------------------------------------- #
    # Time: O(1), Space: O(1)
    def has_solution(self) -> bool:
        return len(self.route) == self.n_cities and
self.distances[self.route[-1]][self.route[0]] < math.inf

    # Time: O(1), Space: O(1)
    def finalize_lower_bound(self) -> None:
        self.lower_bound += self.distances[self.route[-1]][self.route[0]]

    # ------------------------------------------------------------------------- #
    #                                                                           #
    #                                    Heap                                   #
    #                                                                           #
    # ------------------------------------------------------------------------- #
    # Time: O(1), Space: O(1)
    def __lt__(self, other) -> bool:
        return self.lower_bound / len(self.route) < other.lower_bound / len(other.route)

    # ------------------------------------------------------------------------- #
    #                                                                           #
    #                                   Debug                                   #
    #                                                                           #
    # ------------------------------------------------------------------------- #
    # Time: O(n^2), Space: O(n^2)
    def __str__(self) -> str:
        string = ""

        string += "\t"

        for city_index in range(self.n_cities):
            string += str(city_index) + "\t"

        string += "\n"

        for i in range(len(self.distances)):
            string += str(i) + "\t"

            for distance in self.distances[i]:
                string += str(distance) + "\t"

            string += "\n"

        string += "lower bound: " + str(self.lower_bound)

        return string.expandtabs(8)
```