Project 5: TSP Branch & Bound

Zach Eliason

1 (Code)

```
import time
import numpy as np
from TSPClasses import *
from copy import copy
import heapq as hq
import networkx as nx


class SearchState:
    def __init__(self, m, parentBound, i, ncities, visited):
        # Each state will keep track of its current bound cost, matrix, and which cities i
        self.bound = parentBound
        self.matrix = m
        self.visited = visited

        self.visited.append(i)
        self.ncities = ncities

        # current is the current city it's visiting
        self.current = i


    def __lt__(self, other):
        # Override comparison for the priority queue to break ties
        if len(self.visited) != len(other.visited):
            return len(self.visited) < len(other.visited)
        return self.current < other.current
```

```
def reduce(self, m): # Time: O(n^2), Space: O(n)
    # Grab the min values from each row
    # Time: O(n), Space: O(n) since min is already stored in a numpy array
    rows_min = np.min(m, axis=1)[:, None]
    # Replace any occurrences of inf with 0
    # Time: O(n), Space: O(1)
    rows_min[np.isinf(rows_min)] = 0
    # Subtract mins rowwise from matrix
    # Time: O(n^2), Space: O(1)
    m = m - rows_min

    # Grab the min values from each column
    # Time: O(n), Space: O(n) since min is already stored in a numpy array
    cols_min = np.min(m, axis=0)[:, None]
    # Replace any occurrences of inf with 0
    # Time: O(n), Space: O(1)
    cols_min[np.isinf(cols_min)] = 0
    # Subtract mins columnwise from matrix
    # Time: O(n^2), Space: O(1)
    m = m - np.transpose(cols_min)

    bound = sum(cols_min) + sum(rows_min)

    # Return sum of the row and column mins
    return m, bound[0]

def visit(self, m, i, j, current_bound): # Time: O(n^2), Space: O(n)
    if math.isinf(m[i, j]):
        return None, math.inf

    current_bound += m[i, j]

    # Time: O(n), Space: O(1)
    m[:, j] = math.inf
    m[i, :] = math.inf
    m[j, i] = math.inf

    # Time: O(n^2), Space: O(n)
    m, add_bound = self.reduce(m)
    current_bound += add_bound
```

```
        return m, current_bound

    def expand(self): # Time: O(n^3), Space: O(n^3)
        i = self.current

        states = []

        # Time: O(n^3), Space: O(n^3)
        for j in range(self.ncities):
            if j == i:
                continue

            if j == self.visited[0] and len(self.visited) != self.ncities:
                continue

            # Time: O(n^2), Space: O(n)
            new_m, new_bound = self.visit(copy(self.matrix), i, j, self.bound)
            # Space: O(n^2)
            s = SearchState(new_m, new_bound, j, self.ncities, copy(self.visited))
            states.append(s)

        return states


class TSPSolver:
    def __init__( self, gui_view ):
        self._scenario = None
        self.BSSF = None
        self.bssf_cost = math.inf
        self.lower_bound = math.inf

    def init_matrix_no_reduce(self, cities, ncities):  # Time: O(n^2), Space: O(n^2)
        m = np.zeros((ncities, ncities))

        # Time: O(n^2), Space: O(n^2)
        for i in range(len(cities)):
            for j in range(len(cities)):
                # We will treat this as constant since it doesn't increase with n cities
                m[i, j] = cities[i].costTo(cities[j])

        # Time: O(n^2), Space: O(n)
        return m
```

```
def setupWithScenario( self, scenario ):
    self._scenario = scenario

def reduce(self, m): # Time: O(n^2), Space: O(n)
    # Grab the min values from each row
    # Time: O(n), Space: O(n) since min is already stored in a numpy array
    rows_min = np.min(m, axis=1)[:, None]
    # Replace any occurrences of inf with 0
    # Time: O(n), Space: O(1)
    rows_min[np.isinf(rows_min)] = 0
    # Subtract mins rowwise from matrix
    # Time: O(n^2), Space: O(1)
    m = m - rows_min

    # Grab the min values from each column
    # Time: O(n), Space: O(n) since min is already stored in a numpy array
    cols_min = np.min(m, axis=0)[:, None]
    # Replace any occurrences of inf with 0
    # Time: O(n), Space: O(1)
    cols_min[np.isinf(cols_min)] = 0
    # Subtract mins columnwise from matrix
    # Time: O(n^2), Space: O(1)
    m = m - np.transpose(cols_min)

    bound = sum(cols_min) + sum(rows_min)

    # Return sum of the row and column mins
    return m, bound[0]


def defaultRandomTour(self, time_allowance=60.0 ):
    results = {}
    cities = self._scenario.getCities()
    ncities = len(cities)
    foundTour = False
    count = 0
    bssf = None
    start_time = time.time()
    while not foundTour and time.time()-start_time < time_allowance:
        # create a random permutation
        perm = np.random.permutation( ncities )
```

```
            route = []
            # Now build the route using the random permutation
            for i in range( ncities ):
                route.append( cities[ perm[i] ] )
            bssf = TSPSolution(route)
            count += 1
            if bssf.cost < np.inf:
                # Found a valid route
                foundTour = True
        end_time = time.time()
        results['cost'] = bssf.cost if foundTour else math.inf
        results['time'] = end_time - start_time
        results['count'] = count
        results['soln'] = bssf
        results['max'] = None
        results['total'] = None
        results['pruned'] = None
        return results


    def init_matrix(self, cities, ncities): # Time: O(n^2), Space: O(n^2)
        m = np.zeros((ncities, ncities))

        # Time: O(n^2), Space: O(n^2)
        for i in range(len(cities)):
            for j in range(len(cities)):
                # We will treat this as constant since it doesn't increase with n cities
                m[i,j] = cities[i].costTo(cities[j])

        # Time: O(n^2), Space: O(n)
        return self.reduce(m)

    def greedy(self, time_allowance=60.0, init=False): # Time: O(n^4), Space: O(n^2)
        results = {}
        cities = self._scenario.getCities()

        # Time: O(n), Space: O(1)
        ncities = len(cities)
        visited = []

        foundTour = False
```

```python
start_time = time.time()
count = 1

while time.time()-start_time < time_allowance:
    if foundTour:
        break

    # Time: O(n^4), Space: O(n^2)
    for start in reversed(range(ncities)):
        if foundTour:
            break

        visited = []
        i = start

        # Time: O(n^3), Space: O(n^2)
        m, bound = self.init_matrix(cities, ncities)
        while not foundTour:
            if len(visited) == ncities:
                for v in visited:
                    print(v._name, end=" ")
                print()

                foundTour = True
                break

            visited.append(cities[i])

            if sum(m[i, :]==math.inf) == ncities:
                print('fail')
                # FAILURE
                break

            j = np.argmin(m[i])

            # Time: O(n), Space: O(n)
            m[:, j] = math.inf
            m[i, :] = math.inf
            m[j, i] = math.inf

            # Time: O(n^2), Space: O(1)
```

```
                m, bound = self.reduce(m)

                i = j
            break


    # Time: O(n), Space: O(n)
    bssf = TSPSolution(visited)

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

    if not foundTour:
        results = self.defaultRandomTour()

    if init and results['cost'] < self.bssf_cost:
        self.BSSF = bssf
        self.bssf_cost = results['cost']

    return results


def branchAndBound(self, time_allowance=60.0): # Time: O(n^3 * n!), Space: O(n^3 * n!)
    # initialize BSSF
    self.BSSF = None
    self.bssf_cost = math.inf
    self.lower_bound = math.inf
    self.greedy(init=True)

    results = {}
    results['pruned'] = 0
    results['updates'] = 0
    results['total'] = 1
    results['max'] = 0
    solutions = [self.BSSF] if self.BSSF else []
```

```
heap = []

# Time: O(n), Space: O(n)
cities = self._scenario.getCities()
ncities = len(cities)

start_time = time.time()

# Time: O(n^2), Space: O(n^2)
m, parent_bound = self.init_matrix(cities, ncities)
self.lower_bound = parent_bound

# Helps determine searchstate priority in queue
DEPTH_FACTOR = self.lower_bound / (ncities * .9)

start = self.BSSF.route[0]._index
s = SearchState(m, parent_bound, start, ncities, visited=[])
heap.append((parent_bound, parent_bound, s))

# Time: O(n^3 * n!), Space: O(n^3 * n!)
while time.time() - start_time < time_allowance and len(heap) > 0:
    if len(heap) > results['max']:
        results['max'] = len(heap)

    # Time: O(log(n)), Space: O(1)
    s = hq.heappop(heap)[-1]

    # Time: O(n^3), Space: O(n^3)
    states = s.expand()
    results['total'] += len(states)

    # Time: O(n^2), Space: O(n^2)
    for state in states:
        # Skip states that have finished visiting all cities and either can't conn
        # or have a bound of infinity
        if len(state.visited) == ncities and \
        (math.isinf(state.bound) or \
        (state.matrix is not None and \
        math.isinf(state.matrix[state.current, start]))):
            continue
```

```
                if len(state.visited) == ncities and state.bound < math.inf:
                    solutions.append(state)

                    # Time: O(n), Space: O(n)
                    cities_visited = [cities[x] for x in state.visited]

                    if state.bound < self.bssf_cost:
                        # Time: O(n), Space: O(n)
                        self.BSSF = TSPSolution(cities_visited)
                        self.bssf_cost = self.BSSF.cost

                        # Now that cost is updated, prune unneccessary states
                        new_heap = []
                        for x in heap:
                            if x[1] > self.bssf_cost:
                                results['pruned'] += 1
                                continue
                            else:
                                new_heap.append(x)

                        heap = new_heap
                        hq.heapify(heap)

                        results['updates'] += 1

                    if state.bound == self.lower_bound:
                        results['pruned'] += len(heap)
                        heap = []
                        break
                    continue

                if state.bound < self.bssf_cost:
                    # Time: O(log(n)), Space: O(1)
                    hq.heappush(heap, \
                    (state.bound - (DEPTH_FACTOR * len(state.visited)), \
                    state.bound, state))
                else:
                    results['pruned'] += 1

        if time.time() - start_time > time_allowance:
            print('ran out of time!')
            print(f"{len(heap)} more states on the heap")

        end_time = time.time()
        results['cost'] = self.BSSF.cost if self.BSSF else math.inf
        results['time'] = end_time - start_time
        results['count'] = len(solutions)
        results['soln'] = self.BSSF
        results['pruned'] += len(heap)

        return results
```

2 (Performance)

Priority Queue

I used the `heapq` library to implement my priority queue, meaning it has the space and time complexity of a binary heap. Inserting items onto the heap has a time complexity of $O(\log(n!))$ because when it inserts a new item into the binary heap, it needs to then bubble that item up into its appropriate place in the heap. The score of this item will be compared to the scores of up to one node for each of the $O(\log(n!))$ levels of the binary heap. It will swap places with any of these parent nodes which has a greater score than its own. There are $n!$ possible search states that can be stored on the heap. However, in practice there will be nowhere near that number so performance remains acceptable. Removing an item from the heap has a time complexity of $O(\log(n!))$ because when it selects and pops the minimum item in the heap off, it then must replace it with the item at the top of the heap. This item is then bubbled down until its score is greater than its children. The score of this item will be compared to the scores of up to one node for each of the $O(\log(n!))$ levels of the binary heap. It will swap places with any of these child nodes which has a lesser score than its own. This function has a space complexity of $O(n!)$ as this is the maximum number of search states that can be stored in the heap. The actual number will of course be much lower as search states are culled.

When the algorithm finds a new BSSF cost, the priority queue is sifted through, removing any items with higher bounding costs than the new best. This takes potentially $O(n!)$ time, although it happens infrequently and there are never very many items on the queue simultaneously. Whether this functionality falls under the priority queue exactly is up for debate, but if it does than the queue technically operates under $O(n!)$ time and space complexity.

Search States

While there are potentially $n!$ search states, there will be many fewer in actuality. Each one stores a reduced cost matrix, meaning each state has a space complexity of $O(n^2)$. Overall search state space complexity will be $O(n!n^3)$. Each search state will be expanded in $O(n^3)$ space and time. This is because each unfinished state has n possible cities it can visit next. Each of these cities represents a new search state, each storing an $n \times n$ matrix.

Reduced Cost Matrix

Each search state stores an $n \times n$ matrix. Reducing these matrices (as must be done for each new state in a state expansion) takes $O(n^2)$ space and time, as the minimum of each row must be subtracted from the matrix in addition to the minimum of each column. Vectorization using the `numpy` library speeds up this process as the machine doesn't have to check the data type of each object before performing an operation on it, but it doesn't reduce the $O(n^2)$ complexity.

BSSF Initialization

My algorithm is initialized using a greedy approach. It has $O(n^4)$ time complexity and $O(n^2)$ space complexity. This is because it creates an $O(n^2)$ cost matrix of distances from each city to every other city which is reduced (using the $O(n^2)$ complexity described above) for each step in the n steps it takes to find a solution for the TSP. However, this greedy approach sometimes doesn't result in a valid solution, so this $O(n^3)$ is repeated for each of the n cities as starting points, resulting in $O(n^4)$ overall time complexity. Because the greedy algorithm never stores more than a single $n \times n$ matrix at a time, the space complexity remains $O(n^2)$.

Technically, this approach doesn't always result in a valid solution for the hardest setting (Which Dr. Bean said was okay for the greedy) in which case it defaults to the random initialized solution. This approach tried various $n!$ permutations until they work, so the overall time complexity of this greedy algorithm would be $O(n!)$. However, in practice it is much nearer to $O(n^4)$.

Overall

Since there are potentially $n!$ search states, each having $O(n^3)$ space and time complexity, this section of the algorithm complexity dwarfs the BSSF initialization, which as stated above has $O(n^4)$ time and $O(n^2)$ space complexity. The logarithmic operations from the binary heap are similarly dwarfed by the complexity of search state operations. After expanding the lowest cost search state into its n potential new search states, there is the possibility that they will be valid solutions having traversed all n cities. In this case, a solution is computed for each state resulting in $O(n)$ time complexity and constant space complexity. This will be repeated for each of the n new search states, resulting in time complexity of $O(n^2)$ and space complexity of $O(n)$ for this segment of the code. This is still dwarfed by the preceding segment boasting space and time complexity of $O(n!n^3)$, or $O(n^2)$ complexity for each $n!$ state. This will be the overall complexity of the branch and bound algorithm, although in practice the performance will be much more streamlined as search states are filtered out.

3 (Search States)

I wrote a class to represent each new search state object. Each object keeps track of its own reduced cost matrix, current bound cost, an array of visited cities, and the city the state is currently located at.

When called on to expand, it will iterate through the $n-1$ cities not including the current city, and call a visit function on each.

This visit function adds the cost between the current city and the new city to the previous states bound cost as well as the cost returned from the new reduced matrix. It also sets the row, column, and reciprocal coordinates of the new destination to infinity in the new matrix. Following this, the matrix is reduced again and saved. This matrix and the updated bound cost are returned to the expand function.

The expand function then uses this information to generate a new search state. This gets appended to an array of search states which the function then returns to the outer branch and bound algorithm.

4 (Priority Queue)

I used the `heapq` library because I wanted a binary heap implementation of my queue. I already described some of the behavior of the heap earlier in the complexity section. First, my queue ensures that a state's current bounding cost is less than the current BSSF cost. If this isn't the case, the state is thrown out as it can't possibly be lower than the best solution we've got. If it is less, then my queue prioritizes it based on a number of factors: current bounding cost of the state, the number of cities it has visited, the number of cities total (n), and the starting lower cost for the matrix. My exact formula was

$$StateBoundingCost - (\frac{StartingLowerCost}{NumberOfCities \times 0.9} * NumberOfCitiesStateVisited)$$

This computed score would be fed along with the state into the priority queue, where it would be placed according to its score.

When the algorithm finds a new BSSF with a new BSSF cost, the priority queue will be sifted through, removing any items that have costs (not priority scores) higher than the BSSF cost.

5 (BSSF Initialization)

I used a greedy algorithm to initialize the BSSF. It first computes a reduced cost matrix representing how long it takes to get to each city from the others. Then, it chooses one arbitrarily as the origin. Using this starter city, it will then walk through the rest of the cities, choosing the lowest cost city at each step and reducing the cost matrix along the way. If it is able to successfully walk through all possible cities, it has found a valid solution. However, in the hard difficulty there are sometimes gaps between cities where no traversal is possible. This means that sometimes the algorithm will run into a brick wall where it reaches a city that has no way of getting to the remaining unvisited cities. In this case, the algorithm will pick up from the beginning matrix and try again from each of the remaining possible points of origin. This will lead to a valid solution in most cases. On the hardest difficulty, sometimes even cycling through all n possible origins doesn't result in a valid solution. In this case, the greedy algorithm defaults to the random initialization algorithm given to us in the project spec and returns whichever BSSF it was able to find.

6 (Table)

Cities	Seed	Run-time	BSSF (*=optimal)	Max stored states at once	# BSSF updates	Total states created	Total states pruned
10	0	.11s	6421*	75	4	2346	1493
15	1	2.6s	10228*	104	6	43812	39961
20	2	60s	10049	1024	9	1094636	1033697
25	3	60s	11922	649	9	1104278	1056082
30	4	60s	14009	569	7	1033426	996350
50	5	60s	21262	18094	2	1750034	1713527
18	6	12.7s	8846*	152	5	111922	104863
19	7	60s	10416	1501	22	1115134	1048994
20	8	60s	10261	1312	8	1064522	1005256
15	9	1.5s	9118*	142	6	25508	23481
14	1	1.19s	9902*	86	6	20558	18785
14	2	10.9s	9309*	95	2	210902	193315
14	3	.23s	8772	23	1	3314	3038

7 (Results Discussion)

One of the most interesting thing to me about the results table is that the number of states pruned is not proportional to the size of the problem. Whereas my test run of size 10 resulted in approximately 63% of states pruned, when $n = 20$ the percentage pruned increased to 93%. This speaks to the nature of exponential and factorial problems and the way the expand with n. Of course, the absolute number of states increased dramatically as well. Even though at $n = 50$ the algorithm was able to prune 97% of states created, it still had to deal with 18094 states on the stack at one time. With all of the potential states to sift through, the algorithm was only able to update BSSF twice. Of course, the more BSSF updates there are, the more states will be pruned. Ideally both numbers would be relatively high. I also thought it was interesting how different TSP's of the same size could be. For example, my last three runs were all at $n = 14$ but experienced dramatically different performance. This shows that in many cases, the algorithm is able to find a smart bound that quickly reduces the number of options to a manageable number. For that reason I thought the branch and bound algorithm was the most fascinating one we studied in this class.

8 (Strategies Discussion)

As stated above, I determined a score for each SearchState based on the following formula.

$$StateBoundingCost - (\frac{StartingLowerCost}{NumberOfCities \times 0.9} * NumberOfCitiesStateVisited)$$

I settled on this approach after a process of trial and error in which I kept track of the performance of various different strategies and what worked best for most iterations of the problem. In particular, I looked at the number of states created, the maximum length of the priority queue, and most importantly, how many states were filtered. Obviously, I tried to increase the number of filtered states as much as possible, and that performance metric seemed to be the most highly correlated with algorithm speed. In determining my SearchState priority score, *StateBoundingCost* was an easy variable to include; it represented the lowest possible cost for a solution from that state. This meant that if the lowest possible cost was already higher than the best I'd found, we could immediately disregard it. However, I didn't want to simply cycle through every lowest *StateBoundingCost* as this would for the most part run through early SearchStates and never dive deep enough to find a BSSF in the short time allotted. Thus, I started considering the $NumberOfCitiesStateVisited$; by subtracting that value multiplied by some constant, I could give priority to states that were closer to finding a solution because their score would be penalized less for each city they had visited. Having determined this, I only needed to find an optimal constant to multiply $NumberOfCitiesStateVisited$ by. I wanted it to be dependent on each specific TSP, so I took into account *StartingLowerCost* and $NumberOfCities$. This way TSP's with more cities would allow states to dive deeper without resulting in negative scores. I included *StartingLowerCost* in this constant because it was represented in every single state bounding cost and thus, when removed, would show only the differences between state bounding costs. I settled on the .9 value through trial and error. A larger value would prevent states closer to completion from being prioritized, which performed well on smaller TSP's. However, this meant that larger TSP's wouldn't find BSSF's in a timely manner, hurting the algorithm's performance. The .9 constant seemed to perform favorably across the majority of sizes of TSP I tested it on.