CS312 TSP Problem Report

Cannon Farr

1. Code (See Appendix)

2. Time and Space complexity

Greedy Algorithm:

The greedy algorithm works by starting with the first city and going to the next best city for every move. This results in a double-for loop to where for each city, you must compare to the other cities to find the next shortest edge of the unvisited cities. This gives an overall time complexity of $O(n^2)$.

The space complexity for the greedy algorithm is rather simple. In my implementation, I started with a list of cities and moved them from the original list to the route as they were selected. Each city only needed to be stored once, giving a space complexity of O(n).

Priority Queue:

The priority queue is implemented using a binary heap. There are a couple of operations that are performed on the queue:

Insertion: For each insertion, the weight of the path must be compared with the other items in the queue. Given the structure of a binary heap, this takes O(logn) time.

Pop: Similarly, the Pop operation does the opposite of the insertion, and it removes the lowest item in the queue. After the item is removed, the heap must be restructured in order to maintain its integrity. This calculation is also O(logn) time.

The space complexity of the priority queue is simple because it's just the amount of elements that are being stored. There is no extra data that needs to be stored. This gives a Space complexity of O(n).

Reduced Cost Matrix:

For each row and column reduction in an n x n matrix, the smallest element is subtracted from every element in the row or column. This calculation is $2n^2$ giving us a Big-O of $O(n^2)$.

The space complexity for reducing the cost matrix is $O(n^2)$. The operations of both reducing the rows and columns are done on the same n x n matrix. Additionally, we store the value of the reduced cost, but that space is negligible compared to the matrix.

BSSF Initialization:

In my implementation I tried a couple of options for initializing the BSSF, but had the most success with using the result of the greedy algorithm that was implemented earlier. We learned that having the tightest bound on the BSSF would be the most efficient and the greedy algorithm is what gave me that result. For the time and space complexity, it is exactly the same as the greedy algorithm analysis above.

State Expansion:

The state expansion time complexity is a little more complicated because it is dynamic depending on the depth of the route. The further the route is (and the less amount of unvisited cities), there are less possible states that can be branched out of the current one. At the very beginning, the expansion could have a complexity of up to O(n) whereas that value gets smaller as more cities are visited resulting in O(k-n). Additionally on top of the expansions, we have to update and reduce the matrix for each state. Based upon the previous analysis of the matrix reduction, that complexity is $O(n^2)$. Combining these together gives us a final time complexity of $O\big((n-k)*n^2\big)$.

The space complexity is also dynamic on the problem size because for each new state, a new n x n matrix is generated. This gives us the identical space complexity of $O\big((n-k)*n^2\big)$.

Full Algorithm:

In a worst case scenario, we know that if we analyzed every single possible route, that gives an extremely inefficient time complexity of O(n!). The branch and bound algorithm adds the benefit of immediately cutting out solution paths that don't have the potential to be the best solution. The time complexity heavily relies on the amount of child states that are produced and the average rate at which states are pruned. I was able to find a balanced weight between the best lower bound and furthest depth to cut out the worst solutions at an efficient rate. While the time complexity can't be explicitly given, altering the values of the weights is the best way to optimize the time.

In terms of space, again, it all depends on the problem size. The space required for each state is an n x n matrix or $O(n^2)$, and so similar to the time complexity, the more we can optimize the weights, the lower the coefficient will be on the amount of matrices we end up storing at a time.

3. Data structures:

Binary Heap/Queue:

The priority queue is implemented with a binary heap. This queue was full of custom classes used to record the states.

PartialPath class:

I created a custom class to record and represent the individual states of the problem. On this class I recorded the lower bound, the reduced cost matrix, and the weight of its potential to be an optimal solution. I additionally recorded the route used previously to get to that state in a list, and the cost of that route.

4. Priority Queue Implementation:

I used the python library heapq for my priority queue implementation. This mostly helped with the overhead of all of the push methods and rebalancing the tree. It's complexities are described above. The underlying data structure of the queue is just a normal list, but the elements are stored in a way that their indices represent a binary tree. Elements in the tree always have higher values than their parents. The purpose of the heap is that you can pop off the minimum element and the tree will rebalance itself by comparing all its remaining elements.

5. Initial BSSF

For debugging purposes, I initially started with an initial BSSF of infinity. It wasn't as inefficient as I thought it would be because a much lower BSSF would quickly be found, but switching to the result of the greedy algorithm eliminated those extra solutions and allowed more pruning.

6. Table:

# Cities	Seed	Running Time	Cost of Best tour found	Max # of states	# of BSSF updates	Total states	Total pruned	C t
15	20	5.335	9282	35	28	89805	77551	
16	902	48.827	8865	51	46	777521	670533	
20	542	60	11449	55	11	743013	672820	
30	34	60	16455	64	1	490227	463067	
40	257	60	22280	140	0	3455474	342242	
50	931	60	30002	248	0	285639	277321	
10	165	0.112	9485	16	22	2802	2147	
11	189	0.162	6336	20	21	3649	2860	
12	859	0.723	7795	21	18	15106	12373	
13	368	0.119	7719	13	8	2262	1928	

7. Table Results:

The number of states, solutions, and prunes is all dependent on the edges and where the algorithm starts. If the algorithm is able to prune a lot of early inefficient branches, then the algorithm can eliminate very large chunks of the problem early. This allows more specific optimizations that don’t require the big journey of finding a bad solution for nothing.

It was interesting that as the number of cities increased, the number of BSSF updates/solutions decreased significantly. I think this is because at scale, the greedy algorithm produces a pretty good result and ends up being more efficient. If we wanted the branch and bound to make more improvements we’d have to let it run longer than 60 seconds and potentially alter the weights of the queue.

8. Personal Implementation Mechanisms:

There were really only two values that I played with in order to optimize the solution: the initial BSSF, and the weight.

Initial BSSF: See 5.

Weight Formula:

After doing research from the slides and project spec, I figured that having the lower bound be the key to the priority queue would be ideal, because it went to the most potential solutions first. However, this ended up being problematic because final routes were never completed, and it essentially turned into a breadth-first search. I came to the conclusion that I also needed to factor in the route depth and created a formula to measure that.

The weight formula that I used is an inverse proportional relationship of the lower bound and the route depth. Essentially, the weight value for the queue got lower as the lower bound got lower and the route got deeper. Here is my final formula:

$$W = \frac{1}{A \cdot lb + B\left(\frac{1}{depth + C}\right)}$$

The effect of the lower bound and the depth are changed by the constants A and B. I found a 2/3 relationship between lower bound and depth to be a good fit. The constant C is simply a safety net to avoid dividing by 0. I used 1.

Appendix:

TSPClasses.py:

```
#!/usr/bin/python3


import math
import numpy as np
import random
import time



class TSPSolution:
    def __init__( self, listOfCities):
        self.route = listOfCities
        self.cost = self._costOfRoute()

    def _costOfRoute( self ):
        cost = 0
        last = self.route[0]
        for city in self.route[1:]:
            cost += last.costTo(city)
            last = city
        cost += self.route[-1].costTo( self.route[0] )
        return cost

    def enumerateEdges( self ):
        elist = []
        c1 = self.route[0]
        for c2 in self.route[1:]:
            dist = c1.costTo( c2 )
            if dist == np.inf:
                return None
            elist.append( (c1, c2, int(math.ceil(dist))) )
            c1 = c2
        dist = self.route[-1].costTo( self.route[0] )
        if dist == np.inf:
            return None
        elist.append( (self.route[-1], self.route[0], int(math.ceil(dist))) )
        return elist


def nameForInt( num ):
```

```python
    if num == 0:
        return ''
    elif num <= 26:
        return chr( ord('A')+num-1 )
    else:
        return nameForInt((num-1) // 26 ) + nameForInt((num-1)%26+1)



class Scenario:

    HARD_MODE_FRACTION_TO_REMOVE = 0.20 # Remove 20% of the edges

    def __init__( self, city_locations, difficulty, rand_seed ):
        self._difficulty = difficulty

        if difficulty == "Normal" or difficulty == "Hard":
            self._cities = [City( pt.x(), pt.y(), \
                                  random.uniform(0.0,1.0) \
                                ) for pt in city_locations]
        elif difficulty == "Hard (Deterministic)":
            random.seed( rand_seed )
            self._cities = [City( pt.x(), pt.y(), \
                                  random.uniform(0.0,1.0) \
                                ) for pt in city_locations]
        else:
            self._cities = [City( pt.x(), pt.y() ) for pt in city_locations]


        num = 0
        for city in self._cities:
            city.setScenario(self)
            city.setIndexAndName( num, nameForInt( num+1 ) )
            num += 1

        # Assume all edges exists except self-edges
        ncities = len(self._cities)
        self._edge_exists = ( np.ones((ncities,ncities)) - np.diag(
np.ones((ncities)) ) ) > 0

        if difficulty == "Hard":
            self.thinEdges()
        elif difficulty == "Hard (Deterministic)":
            self.thinEdges(deterministic=True)
```

```
    def getCities( self ):
        return self._cities


    def randperm( self, n ):
        perm = np.arange(n)
        for i in range(n):
            randind = random.randint(i,n-1)
            save = perm[i]
            perm[i] = perm[randind]
            perm[randind] = save
        return perm

    def thinEdges( self, deterministic=False ):
        ncities = len(self._cities)
        edge_count = ncities*(ncities-1) # can't have self-edge
        num_to_remove = np.floor(self.HARD_MODE_FRACTION_TO_REMOVE*edge_count)

        can_delete  = self._edge_exists.copy()

        # Set aside a route to ensure at least one tour exists
        route_keep = np.random.permutation( ncities )
        if deterministic:
            route_keep = self.randperm( ncities )
        for i in range(ncities):
            can_delete[route_keep[i],route_keep[(i+1)%ncities]] = False

        # Now remove edges until
        while num_to_remove > 0:
            if deterministic:
                src = random.randint(0,ncities-1)
                dst = random.randint(0,ncities-1)
            else:
                src = np.random.randint(ncities)
                dst = np.random.randint(ncities)
            if self._edge_exists[src,dst] and can_delete[src,dst]:
                self._edge_exists[src,dst] = False
                num_to_remove -= 1



class City:
    def __init__( self, x, y, elevation=0.0 ):
        self._x = x
```

```python
        self._y = y
        self._elevation = elevation
        self._scenario  = None
        self._index = -1
        self._name  = None

    def setIndexAndName( self, index, name ):
        self._index = index
        self._name = name

    def setScenario( self, scenario ):
        self._scenario = scenario

    ''' <summary>
        How much does it cost to get from this city to the destination?
        Note that this is an asymmetric cost function.

        In advanced mode, it returns infinity when there is no connection.
        </summary> '''
    MAP_SCALE = 1000.0
    def costTo( self, other_city ):

        assert( type(other_city) == City )

        # In hard mode, remove edges; this slows down the calculation...
        # Use this in all difficulties, it ensures INF for self-edge
        if not self._scenario._edge_exists[self._index, other_city._index]:
            return np.inf

        # Euclidean Distance
        cost = math.sqrt( (other_city._x - self._x)**2 +
                          (other_city._y - self._y)**2 )

        # For Medium and Hard modes, add in an asymmetric cost (in easy mode it
is zero).
        if not self._scenario._difficulty == 'Easy':
            cost += (other_city._elevation - self._elevation)
            if cost < 0.0:
                cost = 0.0


        return int(math.ceil(cost * self.MAP_SCALE))

    def __eq__(self, __value: object) -> bool:
        return self._index == __value._index
```

```python
class PartialPath:

    # Heap weight constants
    # A controls the weight of the lower bound
    A = 2
    # B controls the weight of the depth
    B = 3
    # C is a constant to prevent division by zero
    C = 1

    def __init__( self, route: np.ndarray[int], matrix: np.ndarray, cost: float,
compute_lower_bound: bool = True):
        self.route = route
        self.cost = cost
        self.matrix = matrix
        self.reduced_matrix = matrix.copy()
        self.lower_bound = None

        if compute_lower_bound:
            self.lower_bound = self.__getLowerBound()
        self.weight = self.__getHeapWeight()

    def __getLowerBound(self):
        reduction_cost = 0
        # reduce rows
        for i in self.reduced_matrix:
            min_val = np.min(i)
            if min_val != np.inf:
                reduction_cost += min_val
                i -= min_val

        # reduce columns
        for i in range(len(self.reduced_matrix)):
            min_val = np.min(self.reduced_matrix[:,i])
            if min_val != np.inf:
                reduction_cost += min_val
                self.reduced_matrix[:,i] -= min_val
        self.lower_bound = self.cost + reduction_cost
        return self.lower_bound

    # Calculate the weight of the PartialPath object for the heap with the
inverse proportion of the lower bound and the depth
    def __getHeapWeight(self):
```

```
        lb = self.lower_bound
        depth = len(self.route)
        weight = 1 / (self.A * lb + self.B * (1 / (depth + self.C)))
        return weight

    # Override the less than operator to compare the weight of the PartialPath
objects
    def __lt__(self, other):
        return self.weight < other.weight
```

TSPSolver.py:

```
#!/usr/bin/python3

from which_pyqt import PYQT_VER
if PYQT_VER == 'PYQT5':
    from PyQt5.QtCore import QLineF, QPointF
elif PYQT_VER == 'PYQT4':
    from PyQt4.QtCore import QLineF, QPointF
elif PYQT_VER == 'PYQT6':
    from PyQt6.QtCore import QLineF, QPointF
else:
    raise Exception('Unsupported Version of PyQt: {}'.format(PYQT_VER))


import time
import numpy as np
from TSPClasses import *
import heapq
import itertools


class TSPSolver:
    def __init__( self, gui_view ):
        self._scenario = None

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


    ''' <summary>
        This is the entry point for the default solver
        which just finds a valid random tour.  Note this could be used to find
your
        initial BSSF.
        </summary>
        <returns>results dictionary for GUI that contains three ints: cost of
solution,
        time spent to find solution, number of permutations tried during search,
the
        solution found, and three null values for fields not used for this
        algorithm</returns>
    '''

    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


    ''' <summary>
        This is the entry point for the greedy solver, which you must implement
for
        the group project (but it is probably a good idea to just do it for the
branch-and
        bound project as a way to get your feet wet).  Note this could be used to
find your
        initial BSSF.
        </summary>
        <returns>results dictionary for GUI that contains three ints: cost of
best solution,
        time spent to find best solution, total number of solutions found, the
best
        solution found, and three null values for fields not used for this
        algorithm</returns>
    '''
```

```python
    def greedy(self, time_allowance=60.0, start_city=0):
        # Initialize variables
        cities = self._scenario.getCities().copy()
        ncities = len(cities)
        start_time = time.time()
        bssf = None
        solvable = True

        #Start with Empty Route
        # Iterate through cities and find the closest city to the last city in
the route
        route = []
        for i in range(ncities):
            cities_copy = cities.copy()
            min_dist = math.inf
            min_city = None
            for j in range(len(cities_copy)):
                # Logic for the first city in the route. It will always be the
start city
                if route == []:
                    route.append(cities_copy[start_city])
                    cities.remove(cities_copy[start_city])
                    break
                # Find the closest city to the last city in the route
                if route[-1].costTo(cities_copy[j]) < min_dist:
                    min_dist = route[-1].costTo(cities_copy[j])
                    min_city = cities_copy[j]
            # If no city is found, the route is not solvable
            if min_city is None:
                if i > 0:
                    solvable = False
                continue
            # Add the closest city to the route and remove it from the cities
list
            route.append(min_city)
            cities.remove(min_city)

        # Create a TSPSolution object from the route
        bssf = TSPSolution(route) if solvable else None
        if solvable is False:
            return self.greedy(time_allowance, start_city+1)
        end_time = time.time()
        results = {}
        results['cost'] = bssf.cost if bssf is not None else math.inf
```

```
        results['time'] = end_time - start_time
        results['count'] = 1
        results['soln'] = bssf
        return results

    ''' <summary>
        This is the entry point for the branch-and-bound algorithm that you will
implement
        </summary>
        <returns>results dictionary for GUI that contains three ints: cost of
best solution,
        time spent to find best solution, total number solutions found during
search (does
        not include the initial BSSF), the best solution found, and three more
ints:
        max queue size, total number of states created, and number of pruned
states.</returns>
    '''

    def branchAndBound( self, time_allowance=60.0 ):
        # Initialize variables
        initial_bssf = self.getInitialBssf()
        bssfCost = initial_bssf
        bssfRoute = []
        cities = self._scenario.getCities().copy()
        start_time = time.time()
        solution_count = 0
        max_queue_size = 0
        total_states_created = 0
        pruned_states = 0

        # Create a PartialPath object with the initial matrix
        initial_matrix = self.getInitialMatrix(cities)
        init = PartialPath([0],initial_matrix,0)

        # Create a priority queue and add the initial PartialPath object
        queue = []
        heapq.heappush(queue, init)

        # Loop through the queue until it is empty or the time allowance is
reached
        while queue:
            if len(queue) > max_queue_size:
                max_queue_size = len(queue)
```

```
            p: PartialPath = heapq.heappop(queue)

            if time.time() - start_time > time_allowance:
                break


            if p.lower_bound < bssfCost:
                t = self.expandAndTest(p)
                for p_i in t:
                    total_states_created += 1

                    # Check if the PartialPath object is a complete route
                    if len(p_i.route) == len(cities):
                        cost_to_start = cities[p_i.route[-1]].costTo(cities[0])
                        # If the cost of the route is less than the current bssf,
update the bssf
                        if p_i.cost + cost_to_start < bssfCost:
                            p_i.cost += cost_to_start
                            bssfCost = p_i.cost
                            bssfRoute = p_i.route
                            solution_count += 1
                    # Otherwise, if the lower bound is less than the current
bssf, add it to the queue for a potential improvement
                    elif p_i.lower_bound < bssfCost:
                        heapq.heappush(queue, p_i)
                    # If the lower bound is greater than the current bssf, prune
the state
                    else:
                        pruned_states += 1

        # Create a TSPSolution object from the route
        bssf = TSPSolution([cities[i] for i in bssfRoute]) if bssfRoute != []
else None

        end_time = time.time()
        results = {}
        results['cost'] = bssfCost
        results['time'] = end_time - start_time
        results['count'] = solution_count
        results['soln'] = bssf
        results['max'] = max_queue_size
        results['total'] = total_states_created
        results['pruned'] = pruned_states
        return results
```

```
    # Create a matrix of costs between cities
    def getInitialMatrix(self, cities):
        matrix = np.full((len(cities), len(cities)), np.inf)
        for i in range(len(cities)):
            for j in range(len(cities)):
                matrix[i,j] = cities[i].costTo(cities[j])
        return matrix

    # Get the initial BSSF. This is configurable to use a greedy tour, infinity,
or a random tour
    def getInitialBssf(self):
        init = self.greedy()['cost'] # Greedy Tour
        # init = np.inf # Infinity
        # init = self.defaultRandomTour()['cost'] # Random Tour
        return init

    # Expand the PartialPath object and create new matrices for each possible
route
    def expandAndTest(self, p: PartialPath):
        exp = []
        for i in range(len(p.matrix)):
            if i not in p.route:
                partial_path_matrix = p.matrix.copy()
                partial_path_matrix[p.route[-1], i] = np.inf
                for j in range(len(partial_path_matrix)):
                    partial_path_matrix[p.route[-1], j] = np.inf
                    partial_path_matrix[j, i] = np.inf
                partial_path = PartialPath(p.route + [i], partial_path_matrix,
p.cost + p.matrix[p.route[-1], i])
                exp.append(partial_path)
        return exp









    ''' <summary>
        This is the entry point for the algorithm you'll write for your group
project.
        </summary>
        <returns>results dictionary for GUI that contains three ints: cost of
best solution,
```

```
        time spent to find best solution, total number of solutions found during
search, the
        best solution found.  You may use the other three field however you like.
        algorithm</returns>
    '''

    def fancy( self,time_allowance=60.0 ):
        pass
```