Code

```
import time
import numpy as np
from TSPClasses import *
import heapq
import itertools
import random
import copy
import math

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>
    '''
    # O(n^3) Time complexity
    # O(n) Space Complexity
    def greedy(self, time_allowance=60.0):
        results = {} # hashmap to store all the results
        bssf = None
        cities = self._scenario.getCities()
        foundTour = False # simple boolean value to keep track if found tour or not
        count = 0
```

```
    start_time = time.time()
    # O(n)
    while not foundTour and time.time()-start_time < time_allowance:
        # O(n)
        for each_city in cities:
            route = self.build_route(each_city, cities)
            if route is not None:
                if bssf is None or route.cost < bssf.cost:
                    count += 1
                    bssf = 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 build_route(self, startCity, cities):
    route = [] # this will keep track of all the routes in order
    visited = [] # keeps track of all the visited cities
    cities_num = len(cities)
    #O (n^2)
    for _ in range(cities_num):
        smallest = math.inf
        nextBestCity = None
        for city in cities:
            if city not in visited:
                length = startCity.costTo(city)
                if smallest > length:
                    smallest = length
                    nextBestCity = city
        nextCity = nextBestCity

        if nextCity is None:
            return None
```

```
            route.append(nextCity)
            visited.append(nextCity)
            startCity = nextCity

        return TSPSolution(route)

    ''' <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 ):
        results = {} # hashmap to store results to pull from
        maxQueue = 0
        totalStatesCreated = 0
        totalStatesPruned = 0
        count = 0 # number of solutions found

        bssf = self.greedy()['soln'] # using solution from greedy to find initial BSSF
instead of using default tour
        cities = self._scenario.getCities()
        pqueue = [] # priority queue
        heapq.heapify(pqueue) # heapify priority queue

        matrix = [[math.inf for _ in range(len(cities))] for _ in range(len(cities))]
        # print(matrix[0][0])
        # print(matrix[-1][-1])

        for i in range(len(cities)):
            for j in range(len(cities)):
                matrix[i][j] = cities[i].costTo(cities[j])

        # print(matrix[0][0])
        # print(matrix[-1][-1])
```

```
        index = random.randint(0, len(cities) - 1) # random starting point

        # We need to create an initial state and start reducing it first
        initialState = self.createInitialState(matrix, cities, index)
        #print(initialState)

        heapq.heappush(pqueue, initialState)

        start_time = time.time()
        while len(pqueue) > 0 and time.time() - start_time < time_allowance:
            currentState = heapq.heappop(pqueue) # getting the top current state off
the queue

            if len(pqueue) > maxQueue: # updating the siZe of maxQueue
                maxQueue = len(pqueue)

            if currentState.lower_bound < bssf.cost:
                if len(currentState.visited_route) == len(cities): # if we have
visited every single city
                    last_city = currentState.route[-1]
                    start_city = currentState.route[0] # initial city with first
initial random index each time arbitrary value

                    if last_city.costTo(start_city) is not math.inf:
                        solution = TSPSolution(currentState.route) # give current
array of routes

                        if solution.cost < bssf.cost: # if the solution cost is
actually less, substitute
                            bssf = solution

                            # Pruning - O(n) operation
                            for i in pqueue:
                                if currentState.lower_bound >= bssf.cost:
                                    pqueue.remove(i)
                                    totalStatesPruned += 1
                            count += 1 # increment number of solutions found
                    else:
                        continue
                else:
                    for i in range(len(cities)):
                        if i not in currentState.visited_route:
```

```
                            newState = States(currentState, i, cities) # generate a
new state
                            totalStatesCreated += 1 # increment number of new states
created

                            if newState.lower_bound < bssf.cost and
newState.lower_bound != math.inf:
                                heapq.heappush(pqueue, newState)
                            else:
                                totalStatesPruned += 1 # TODO: Double check if this is
where I was missing pruning

        end_time = time.time()

        # Results map
        results['cost'] = bssf.cost
        results['time'] = end_time - start_time
        results['count'] = count
        results['soln'] = bssf
        results['max'] = maxQueue
        results['total'] = totalStatesCreated
        results['pruned'] = totalStatesPruned

        return results

    # O(n^2)
    def createInitialState(self, matrix, cities, startIndex):
        cost = 0
        depth = 1
        startIndex = startIndex

        # Creating the initial state, reducing it and then returning the result
        initialState = State(None, None, None) # Try it with different values and see
what has changed
        initialState.set_first_state(matrix, cities, startIndex, cost, depth) # Try it
with different values and see what has changed

        return initialState

    def generateStates(self, currentState): # Is there a better way to handle states?
        pass
```

```
    ''' <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

class States:
    def __init__(self, current_state, city_index, cities):
        self.current_state = current_state # Needs to keep track in which state we're
currently working on
        self.matrix = copy.deepcopy(current_state.matrix) # Copy current State matrix,
but it might slow down a bit
        self.depth = self.current_state.depth + 1 # Keep track of depth
        self.lower_bound = 0 #InitialiZe and keep track of lower bound
        self.parent_state_lower_bound = self.current_state.lower_bound # Keep track of
the parent's lower bound, might not need it we'll see
        self.rows = copy.deepcopy(current_state.rows) # copy rows that will be used to
be reduced each time
        self.cols = copy.deepcopy(current_state.cols) # copy cols that will be used to
be reduced each time
        self.route = copy.deepcopy(current_state.route) # copy routes
        self.visited_route = copy.deepcopy(current_state.visited_route) # copy visited
routes each time
        self.from_city_index = current_state.to_city_index
        self.to_city_index= city_index
        self.reduce_matrix(cities) # Reduce the matrix

    def __lt__(self, other): # Someone in Slack said this might help speed things up?
        return True

    # function to reduce further states
    def reduce_matrix(self, cities):
        cost_of_path = self.matrix[self.from_city_index][self.to_city_index] # Keeping
track of the cost from a city to another city
```

```
        for i in range(len(self.matrix)): # blocking out row with inf
            self.matrix[self.from_city_index][i] = math.inf

        for i in range(len(self.matrix)): # blocking out cols with inf
            self.matrix[i][self.to_city_index] = math.inf

        self.matrix[self.to_city_index][self.from_city_index] = math.inf # get
speficic index from city to city and mark as inf
        self.rows.add(self.from_city_index) # keeping track of row value to not
accidentally reduce it
        self.cols.add(self.to_city_index) # keeping track of col value to not
accidentally reduce it
        cost = self.reduce_rows() # cost of reducing rows
        cost += self.reduce_cols(self.to_city_index) # cost of reducing add
        self.lower_bound = cost + cost_of_path + self.parent_state_lower_bound #
adding costs to lower bound together
        self.route.append(cities[self.to_city_index]) # add current city to the routes
array and to visited set
        self.visited_route.add(self.to_city_index)

    def _reduce(self): # this will only reduce the initial state without making a copy
just yet
        cost_row = self.reduce_rows()
        cost_col = self.reduce_cols(self.to_city_index)
        self.lower_bound = cost_row + cost_col

    def set_first_state(self, matrix, cities, startIndex, cost, depth):
        self.matrix = copy.deepcopy(matrix)
        self.parent_state_lower_bound = cost
        self.depth = depth
        self.visited_route = set()  # keep track of the cities that we have visited
        self.route = []
        self.cols = set() # keep track of the cols and rows that are marked as inf
        self.rows = set()
        self.to_city_index= startIndex # starting city index (random in this case)
        self.route.append(cities[startIndex]) # add the starting index to visited
route
        self.visited_route.add(startIndex) # add the starting index to visited route
        self._reduce() # Initial reduce for the first State - only needed once

    def reduce_rows(self):
```

```
        cost= 0
        for row in range(len(self.matrix)):
            if row not in self.rows:
                matrix_row = self.matrix[row] # getting the minimun value present in a
row
                smallest_value = math.inf
                for i in range(len(matrix_row)):
                    num = matrix_row[i]
                    if num == 0:
                        smallest_value =  num
                        break
                    elif num < smallest_value:
                        smallest_value = num
                if smallest_value > 0 and smallest_value != math.inf:
                    for col in range(len(self.matrix)):
                        cur_val = self.matrix[row][col]
                        self.matrix[row][col] = cur_val - smallest_value
                    cost += smallest_value
        return cost

    def reduce_cols(self, colTo = 0):
        cost_reduce = 0
        min_bool = True # making sure we haven't already updated the minimun value
when updating the row
        for row in range(len(self.matrix)):
            if colTo == 0:
                min_bool = True # only if we reduce the initial state
            elif row in self.cols or row is colTo: # this will guarantee that we
haven't actually visited before
                min_bool = False
            if min_bool: # If we haven't updated a column value when updating the rows
                smallest_value = math.inf
                for i in range(len(self.matrix)):
                    num = self.matrix[i][row]
                    if num == 0:
                        smallest_value =  num
                        break
                    elif num < smallest_value:
                        smallest_value = num
                if smallest_value > 0 and smallest_value != math.inf:
                    for col in range(len(self.matrix)):
                        cur_val = self.matrix[col][row]
```

```
                    self.matrix[col][row] = cur_val - smallest_value
                cost_reduce += smallest_value
        min_bool = True
    return cost_reduce
```

Time and Space Complexity Analysis

Priority Queue:

Heapify - Since we're converting it into a heap data structure, in the worst case scenario this will have an O(n) time complexity and O(n) space complexity.

Push and Pop operations: Both run in O(logn) time and O(1) space complexity.

def reduce_matrix(self, cities):

Creating and updating the states runs in O(n^2) time because we're looking at each individual cell in order to calculate the lower bound. This is done by first reducing the rows, which calls the reduce_rows function, and then it calls the reduce_cols function to reduce the cols. This operation has O(n) time complexity because it creates and uses a copy of the parent's state, and then substitutes it as the new current state.

def reduce_rows(self):

In order to reduce the rows, this function runs in O(n^2) because it loops through all the rows, and then looks at it again in order to find and get the smallest present value. This doesn't generate any more space, so it has a O(1) time complexity.

def reduce_cols(self):

In order to reduce the cols, this function operates very similarly to the reduce_rows, so it also runs in O(n^2) and takes O(1) space.

BSSF Initialization using def greedy(self, time_allowance = 60.0):

The entire function runs in O(n^3) time and takes O(n) space. The main reason for that is the need to loop through the city multiple times in order to build the route's path that will then be passed to the TSPSolution in order to calculate the bssf. I created a helper function called build_route that does that and returns the TSPSolution based on the route found.

def build_route(self, startCity, cities):

This function runs in O(n^2) and takes O(n) space because it keeps track of the visited cities and route. It will take care of calculating the cost to a city and finding the smallest value possible, and then it will store that in the route and visited arrays, and continue on to find another city. It runs in O(n^2) because we are constantly checking the city we're at and comparing the cost to get to all the next cities, and returning whichever costs the least amount.

def branchAndBound(self, time_allowance = 60.0):

The full Branch and Bound algorithm run in O(k*n^2), where k is the number of states. We take O(n^2) time and space in order to create the full matrix (2D array), as well as O(n^2) for finding the initial reduced matrix and its lowest bound. Summing all the operations that take place in the algorithm, we end up with an algorithm that runs in no more than O(k * n^2).

Data Structure for States

In order to represent the states, I used a simple built-in 2DArray. I used a set to keep track of both the rows and cols that have already been reduced. I also stored the current route in a list and the visited cities in a set. I also stored the lower bound in a simple variable. I created a separate class that helped me keep track of all this information and was updated dynamically everytime I generated a new state - this way it was easier to make copies of the current state, keep track of the number of states that were created and pruned as well.

Priority Queue

I used the priority queue to sort and to get the current states off the top of the queue in order to calculate the cost of the lower bound. Each time I pop the current state, I compare it with the current BSSF, either come up with a better solution or remove the state from the queue and count it as a pruned state.

Initial BSSF Approach

Instead of using the random tour, I decided to code up the greedy algorithm in order to find the initial bssf. Because it's somewhat trivial and it runs fast, it does not add any significant time to my final solution because it is able to get the initial bssf cost quite quickly. In order to develop the greedy algorithm, I simply took the default random tour example and instead of building a random route using random permutations, I built a route based on the minimum cost to get to the next city, and appended the routes in order.

Table and Results

# Cities	Seed	Running Time (sec.)	Cost of best tour found (*= optimal)	Max # of stored states at a given time	# of BSSF updated	Total # of states created	Total # of states pruned
14	1	6.82	10627*	119	6	6455	5514
15	20	5.31	9316*	72	3	4026	3496
16	760	12.75	11457*	251	8	9063	7898
16	902	51.41	8433*	339	4	36218	31625
20	135	60	13823	2818	0	30118	24586
22	158	60	13594	867	0	24964	22004
12	430	4.7	8192*	52	1	4558	3779
35	994	60	21833	2146	9	12157	9481
19	571	60	14573	1465	3	30004	26202
17	16	57.3	10876*	613	9	32579	28383

It's very clear that with the branch and bound algorithm, as the number of cities increase, the number of states also increase, which are then multiplied into the entire algorithm, making it so the larger number of cities time out at 60 seconds. I think that without the greedy implementation in order to find the initial bssf, the algorithm itself may take even longer for the same number of tested cities. Something obvious to note as well, is that as the number of cities increase, the number of states created and pruned increase by a lot as well. Although, with a larger number of cities such as 35, I didn't get nearly as many states created or pruned, but if there wasn't a time limit, the numbers would certainly be very large.

State Space Search Mechanism and Final Observations

Some of the mechanism I put in place which I thought could be helpful was keeping track of the columns and rows that were marked as infinite already so that when we loop over again in order to further reduce the matrix we would check to see if the columns or rows we're at were already taken care of. I'm not too sure how helpful it was time wise, and it definitely did add another layer of complexity when writing the algorithm - however, if all the rows or columns have already been marked then this would potentially save some time. Implementing the greedy solution in order to find the bssf also aids the efficiency of the algorithm, although I think I could have handled it a bit better in making sure that there is always a solution. I think that sometimes it simply returns infinity, and I could have potentially made sure that it's not always going to be infinity and that might speed up the branch and bound algorithm itself in doing so as well. Something else that I thought could have been helpful was creating a separate class to handle all the state operations and making copies of the current state before reducing them. What I hoped to accomplish is to make it easier to manage the states, but I've noticed that doing so really slowed down some of the operations. If I had more time, I would have tried to figure out another way to more efficiently dig further into the states and more efficiently reduce the matrix, which could potentially lead it to not timing out in some of the larger operations.