Gregory Knapp

CS 312

Dr. Grimsman

11/29/2021

Project 5: Traveling Salesperson Branch and Bound.

# Time Complexity Analysis of Project 5

The time complexity for each part of the code is pulled from the source code comments, with any additional notes added here for things that aren’t explained within the code.

Analysis of Priority Queue

TSPSolver.py Lines 17 - 31

# Binary Heap implementation of a Priority Queue. The code is essentially the same from my project 3, with a few  
# specializations to make it work for this project (using a unique priority key instead of values from dist array and  
# other changes. These changes don't affect the performance of this data structure. The time complexity of  
# each function is detailed before, with insert, decrease\_key, and delete\_min all performing at a time complexity  
# of O(log n) because they need to sort the values up or down depending on the function, which takes about log n  
# iterations to complete on average or worst case.  
#  
# Unlike the previous project, the space complexity of the heap queue is more interesting than before. Because of the  
# way the branch and bound function trims and prunes states, you do not necessarily get a heap queue with O(2^n)  
# exponential side because not every state is added to the queue. It is entirely dependent on the problem to  
# solve, but when testing my code, a TSP problem with 15 or less cities rarely had a max queue size of  
# more than 100. For a 10 city problem, 100 max queue size would be a size complexity of O(n^2), so that is a good  
# starting evaluation of space complexity for the heap queue in this context. There were always outliers in testing  
# for max queue size, but in general, it was very rare to see a max queue size greater than O(n^2) where n is the  
# number of cities to visit, making O(n^2) a safe upper bound on space complexity.

Analysis of Search States

TSPSolver.py Lines 234 – 249

# Node is the class used to represent search states in this program. A search state Node has 3 main arrays; the  
# cost matrix array that contains the current values for each possible path as they are updated, the path array which  
# contains the actual city objects of each city that has been visited that is used by TSPSolution to actually generate  
# a solution object, and the cities\_visited array which stores the index of the cities that have been visited so far.  
# The cities\_visited array is not needed for creating the end solution, but since the cost matrix operates in terms of  
# source/destination path pairs, it was a lot easier to the index of each city ready to go instead of getting them from  
# the cities array every time, even at the cost of additional space to store the indices.  
#  
# Inside of the node class, there are functions to calculate the cost matrix for a city visited, as well as to check  
# and see if a path is a valid solution. The node.lowerbound stores the bound/cost of the current path, and is  
# referenced often through the class and branch and bound solution to check for pruning and binding of states.  
#  
# Time and space complexity of each individual function is given in the comments above each one. In general however,  
# functions that solve/reduce the cost matrix are of time complexity O(n^2) since they are iterating over the complete  
# n x n array. The overall space complexity of each node is dictated by the same n x n or O(n^2) size array that  
# is used for the cost matrix.

Analysis of Reduced Cost Matrix

TSPSolver.py Lines 290 – 303 – generateCostMatrix function

# A function that is only used once by the initial state to generate the numpy array used as the cost matrix.  
# This function is not called when creating children states, a deep copy is made by the init function to pass on  
# a new array object in the same current state. This function builds the array from the list of cities passed on  
# from the GUI. The cities are added in the order they are found in the cities list. After creating the array,  
# the cost of each path to and from each city is added to the array at the indices of each source/destination  
# path combination.  
#  
# The array generation itself is an O(n^2) time complexity since it initializes the array of that size with zeroes  
# for each value. Aside from just the array generation, the function then loops through each row and column of the  
# array to check and update the value based on the scenario, which is also a time O(n^2) section of code, since the  
# array has rows and columns equal to the number of cities to visit.  
#  
# Similar to time complexity, the array generated is an n x n array where both length and width are the number  
# of cities to visit, making the space complexity of this function O(n^2)

TSPSolver.py Lines 322 – 330 – reduceMatrix function

# reduceMatrix is nearly identical in function to the get\_initial\_lower\_bound function below. Both of them  
# center around the behavior of getting the minimum values for the rows and columns in order to reduce the cost  
# matrix for a child state. However, the child states also need to account for the current path being taken as  
# well as account for the cost of the parent path up to this point, making it easier to have the two functions be  
# separate.  
#  
# Although there are a couple additional parts to this function, there is no difference in overall time complexity  
# compared to get\_initial\_lower\_bound, since the nested for loops still dictate the time complexity of this  
# function. Both times, every value is iterated over and updated if needed, making time complexity here O(n^2)

TSPSolver.py Lines 393 – 405 – get\_initial\_lower\_bound function

# The initial function used by the first state created in the branch and bound method to get the starting  
# lower bound value for the rest of the children state to use in their calculations. Functions is extremely  
# similar to reduceMatrix function above, and likely could be combined, but because the cost of the lowerbound  
# of the children nodes rely on the initial lowerbound, the calculations are slightly different, so I ended up  
# having the two functions split since they were developed that way.  
#  
# The cose uses numpy.amin to extract the minimun values from each row or column, then iterates over the  
# entire array to update the values as needed (if they are not infinity or 0). The rows are handled first, since  
# updating the rows may change if there is a need to reduce the column values. For either rows or columns, every  
# value has to be checked and updated, so each nested for loop has a time complexity of O(n^2), since the cost  
# matrix itself has a space complexity of O(n^2), where n is the number of cities to try and visit. Even though  
# there are two sets of nested for loops, the overall time complexity of this function is O(n^2), since every  
# value of the array has to be checked.

Analysis of BSSF Initialization

TSPSolver.py Lines 440 – 448 – Node.get\_greedy\_path function

# The main function used by the greedy algorithm to calculate what is the shortest path from the current  
# city to any other city in the array. It is a greedy function, because it simply chooses the best path at  
# the time, even if the final solution after that choice may not exist. It is only concerned with getting the  
# lowest cost path from this state.  
#  
# Every time this function is called, it iterates over the list of cities. It does skip over iterations where  
# the city has already been visited, but that does not change the fact that the for loop still iterates for that  
# city to start. This for loop over the array of cities makes the time complexity of this function O(n), where n  
# is the number of cities to visit.

TSPSolver.py Lines 573 – 594 – TSPSolver.greedy function

# A greedy algorithm implementation of the TSP solver. While intended for Project 6, I decided to make it earlier  
# to use it to get an initial bssf state for the branch and bound function, since it would get me a decent estimate  
# resulting in more states pruned earlier and less states to create as a result.  
#  
# The function works as described in the specs; it starts from each city and checks the values for all possible  
# paths from the current city to any city not visited and selects the smallest one, adding it to the current path.  
# If it ever encounters a minimum value of infinity, the path is deemed impossible and no solution is added.  
# The function tries this greedy approach for every starting city, updating the best solution from each of the  
# starting points, returning what ever is the best solution.  
#  
# The greedy approach has a time complexity of O(n^3) since there are essentially three nested loops going on.  
# The for loop that determines the starting city iterates over the length of the cities array, giving us O(n) time  
# for the outer-most loop. Inside, there is a while loop that calls the get\_greedy\_path function until the path  
# is complete (len(path) == len(cities)). While this while loop will break early if the algorithm encounters an  
# invalid solution, on average it will run through the length of the cities array to get a solution, giving us  
# another O(n) time loop. Lastly, the call to get\_greedy\_path is the third nested loop, since the function itself  
# loops over the length of the cities array to check for the minimum value of the paths out of the current city,  
# resulting in the third O(n) time complexity loop. For a perfect array that has all valid solutions for each  
# starting city, that would give us O(n^3) time complexity as an upper bound for time complexity.  
#  
# Space complexity for the greedy solution is simply O(n^2), since we are not creating many states like in the  
# branch and bound method, just one that has a n x n array for the cost matrix that is referenced the entire time.

Analysis of Expand function

TSPSolver.py Lines 673 – 693 – TSPSolver.expand function

# The function used by the branch and bound method to expand a node into child states for each possible path  
# out of the current city. The current state is passed in and its values are used to create the new states since  
# the node.\_\_init\_\_ function copies the arrays of the parent node. Expand iterates over the list of cities,  
# checks that the current city is not already on the path of the node. If it is, the city is skipped. If the  
# city has not been visited yet, then a new state is created using the city, which is added to the end of the path.  
# The new state is then passed into the result array T, which is ultimately returned to the main branch and bound  
# method.  
#  
# The time complexity of this function is O(n^2) because of the node creation process. The expand function itself  
# is simply and only relies on a for loop to iterate over the list of cities and create nodes for cities that  
# have not been visited, making it a O(n) function. However, the init function for creating a new node is O(n^2)  
# since the n x n cost matrix has to be copied from the parent state to the new child. This is handled with  
# copy.deepcopy(), which ensures there is not a just a copying of reference to the array, but adds time to the  
# process. The call to reduceMatrix is also O(n^2), ensuring the time complexity of the function to be O(n^2), not  
# O(n)  
#  
# The space complexity of this function is more complicated. Each node created is of size O(n^2) since it contains  
# a unique n x n array for the cost matrix of this unique state. As the states get deeper, less nodes are created  
# per call to expand, but as many as n-1 nodes can be created when expand is called. These nodes are stored in  
# a result array, T, which has n-1 size at its biggest. This makes the space complexity of this function to be  
# O(n^3), since almost n nodes of size n^2 can be created in one call.

Analysis of Branch and Bound Full Algorithm

TSPSolver.py Lines 716 – 750

# This is the function for the core branch and bound algorithm. The initial state is created from the array of  
# cities from the scenario given by the GUI. The branch and bound solution is always started from city at index 0.  
# After the initial state's cost matrix and path is handled and the starting lower bound is found, the heap queue  
# is created and an initial bssf using the greedy algorithm is found. From there, the looping begins;  
# for each iteration, the state with the lowest priority key is popped from the heap and expanded to child states.  
# Each of the child states is check for a valid solution. If a solution is found, then the cost is compared to the  
# bssf. If the cost is an improvement, the solution is added. If the current state is not a complete solution, it  
# is checked against the bound for pruning. If the current state is under the bound, it is pushed onto the heap,  
# else nothing happens and the state is pruned. Each iteration of the while loop checks the elapsed time, breaking  
# the loop at its head if the expected time allowance is passed.  
#  
# The time complexity of the function is still the exponential time of O(2^n). The branching and bounding solution  
# ensures that we get closer to a solution by not spending as much time working on states that do not go anywhere.  
# That does not change the fact that we create all of the states using the expand function in order to check them  
# for pruning. Each state results in an exponentially growing number of children states until a complete solution  
# is found. The pruning reduces the number of states created, especially when a state exceeds the bound early into  
# the path. The greedy algorithm returns a fairly lower bssf, which results in slightly higher early pruning  
# than compared to when I ran branch and bound with the random solution. Still, as the total number of states  
# created in my testing indicates, the number of states created is still an exponential number, making the while  
# loop continue for an exponential time. Of course in our code, we use a time limit to stop the program after  
# a max of one minute, but if the code were to continue, the run time would take a while since it has to loop  
# until there are no more states left on the queue to loop through. Since every state create is a loop on the for  
# loop, the time complexity is O(2^n)  
#  
# The space complexity for branch and bound is a bit more complicated. While the time is exponential, the space  
# complexity of the function is not because of the pruning used. As mentioned in the comment above the heap queue,  
# in testing, it was rare to see a heap queue max size result greater than O(n^2), with n being the number of  
# cities to visit. Since the nodes contain the n x n cost matrix and each have a space of O(n^2), then we can  
# say the heap queue has a space complexity of O(n^4), assuming the upper bound of the queue size is O(n^2).  
# The nodes are still created by the expand function, which is as mentioned in the comments there,  
# space complexity of O(n^3) since the nodes have a n x n array, and up to n - 1 nodes are created. If we consider  
# the heap queue to have a space complexity upper bound of O(n^2) based on the max queue size results, and the  
# expand function to return an array T of size O(n) containing nodes of size O(n^2), then our space complexity  
# of the branch and bound function would be O(n^4), since the O(n^4) size of the heap queue with nodes at max size  
# would dominate the O(n^3) size of the T array containing nodes created by expand.

# Explanation of Node class for Search States

The Node class found in TSPSolver.py is used to represent the search states as described in the comments included in the time/space analysis section of the report. A node has 4 main objects, 3 arrays and an integer to store the lower bound. One of the arrays is the cost matrix. Since each state created has a unique cost matrix based on the path taken, this matrix is copied from the parent state when the node is created using deep copy to ensure that the new matrix is not a reference to the previous one since the path is different now. The other two arrays are arrays to keep track of the both the actual city objects that make up the path, and the index of each city so far in the path according to the cost matrix. The city object array (path array) is necessary since the TSPSolution constructor uses it to generate a solution that can be returned to the GUI, the cities\_visited array is added just for convenience. Since the cost matrix is indexed by source city / destination city, it is helpful to have number index for a city readily available to use for manipulating the cost matrix.

When a state is expanded in the expand function, the arrays and lower bound from the parent node used to create each new child node are passed to the \_\_init\_\_ function to make sure the child starts off as a perfect copy of the parent as a unique object, not a reference to the parent. Immediately after, the cost matrix is reduced based on the city the state represents, and from there the node is used by the branch and bound function.

As described in the comments, the resulting Node structure has a size of about O(n^2), as the n-by-n size cost matrix dominates the size complexity. All of the functions needed to update the cost matrix or check for a solution are found inside of the node class and are called where needed by the branch and bound function.

# Explanation of Priority Queue Implementation

For Project 5, I used my heap q implementation from Project 3 to store the nodes. I did not actually realize that the default heapq was imported for the project and was an option, but I likely would have decided to use my implementation anyways since it is easier to edit the functionality that way.

Not much changed from the original implementation from before, the queue used an insert, decrease\_key, and delete\_min function to handle queue operations. There was a make\_queue function before, but I just reworked the functionality to the \_\_init\_\_ function and started with insert directly just to simplify things.

The time and space complexity of the heap queue is the same before, with all three functions having a time complexity of O(log n) since they must percolate the nodes up and down the heap until the heap is valid. All those features from the heap queue carried over from the previous project.

The big difference in the heap queue this time was the way the priority key was calculated. In Project 3, I simply used the distance from the array of distances and prioritized the lower distance. For this queue, I started with a queue that prioritized the node with the smaller lower bound/path cost. I thought that it would be best to just focus on whichever state seemed the most promising, assuming the lowest cost would create the most children states and have the best chance at a valid solution.

After reading the lab specs more, I realized that just having the lowest cost is not the only factor to consider. While it may help to get a better initial solution, just prioritizing the lower bound would make the algorithm take longer to update the BSSF. In order to counter this, I tried to add in the current length of the path as a factor to consider with the priority key. I created a priority variable that equaled node.lowerbound / len(node.path). The thought behind this was if there were two paths with the same lower bound, the longer path would have a lower priority, and thus be placed higher on the heap. The result did make a difference. Tests run after had an increase in solutions found and pruned states as a result of the more frequent updates to BSSF.

# Explanation for BSSF Initialization

My first instinct was to simply use the provided random solver to get any solution for the BSSF, since anything was better than starting with infinity and pruning no nodes until the first solution is found. However, as should be obvious, using the random solution has some obvious disadvantages, mainly in the form of extremely inconsistent test results. When the random solver get a decently low solution for the initial result, the branch and bound pruned many states quickly and found less solutions. While that is a nice result speed-wise, it is brought down by the tests that return an invalid path for the random solution and left me with pretty much nothing to work with.

After that I thought that maybe a greedy algorithm would give a decent BSSF without likely getting the ideal solution. Since I needed a greedy TSP solver for Project 6, I figured it wouldn’t hurt to use it as part of Project 5. For a problem of size 14 one deterministic hard for seed 1, the greedy algorithm returned a path of cost 11478. My branch and bound algorithm for the same problem found 10 solutions, with the result costing 10573, and having 30481/36348 states pruned (see results table below).

From this and other tests, I found that the greedy algorithm generally gave a decently strong and low estimate for the initial BSSF which resulted in a lower total state count and a higher ratio of pruned states since the initial bound was fairly strong, without being too strict. Especially when compared to the random solution, the greedy solution felt like a much stronger option for a starting solution for the initial BSSF.

# Test Results Table

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| # Cities | Seed | Run Time (second) | Cost of Best Tour \*Optimal | Max # of stored states | Total # of BSSF updates | Total # of states created | Total # of states pruned |
| 15 | 20 | 52.225208 | 10534 | 177 | 17276 | 212008 | 118320 |
| 16 | 902 | 9.455963 | 7954 | 80 | 8 | 32046 | 28292 |
| 10 | 279 | 1.081894 | 8964\* | 34 | 680 | 8964 | 4625 |
| 14 | 82 | 2.588001 | 8812 | 54 | 41 | 10674 | 8912 |
| 15 | 404 | 38.088855 | 8787 | 89 | 10 | 145710 | 124911 |
| 20 | 541 | 60.002678 | 11186 | 125 | 8 | 135527 | 122388 |
| 30 | 11 | 60.00 | 14985 | 329 | 10 | 70139 | 60500 |
| 40 | 65 | 60.001550 | 16886 | 662 | 13 | 42654 | 35020 |
| 50 | 648 | 60.000396 | 22274 | 1576 | 2 | 28006 | 20219 |

# Discussion of Table Results

For the most part, the table results lined up with my expectations based on knowledge of what the algorithm should do.

One first thing that was surprising from time to time was the uncontrollable nature of TSP Problems that sometimes breaks expectations for time complexity. The best example is the contrast between rows 1 and 2 of the table. Row 1 is for a 15-city test on seed 20, while row 2 is the 16-city test on seed 902. The larger the problem gets, the longer it should take since the algorithm is of exponential time complexity. That generally held up true for the tests on the table, but in this case, the 15-city problem took significantly longer than the 16 city one. Comparing max states stored, number of solutions, states created, and states pruned, it’s clear that the 15-city problem was much more complex to figure out and the 16 city one. It just proved that despite certain expectations regarding time complexity, sometimes outlier problems can make things much more difficult. Likely in this case, there were many nodes that has a lower bound very close to the optimal solution, making it hard to prune states without a very aggressive initial BSSF, and as a result calculation took a lot longer and the BSSF was updated many more times.

Another trend I noticed from the table was the generally high rate of states pruned. While a couple of outliers had closer to a 50% pruning rate, many of them were closer to 80% or even 90%. I would attribute this result to two things; the adjustment to use path length to prioritize paths near completion to update BSSF more often in the heap queue, and the generally low initial solution found by the greedy algorithm. I think the greedy algorithms solution is noticeable especially in the number of updates to BSSF. When the count of updates is lower for a solution, it was when the greedy algorithm got a good result to start. Given the nature of greedy algorithms, there’s no way to ensure that the initial solution is very good, but when it found a good initial result, the effect was clear in the number of BSSF updates, pruned states, and run time.

The last main trend I noticed was the growth in space complexity with problem size. Most of the problems of size 10-15 had a similar max state stored, ranging between 30 – 80 (except for the first row, which has already been discussed). From there, the number of max states in the queue began to grow very quickly with 125 for a problem of size 20, 329 for a size 30 problem, and a max of 1576 for a size 50 problem. From a logical perspective it makes sense; pruning really starts to pick up after the first solutions are found, and the larger the problem size, the longer it will take to find a solution beside the initial one. In this regard, the space complexity of the problems increased in a way that lined up with my expectations based on my analysis of the algorithm.

# Discussion of Mechanics to Find Solutions Quickly

The two main mechanics I used to try and get solutions quickly were the priority queue and the initial BSSF solution which are also described in the Explanation sections of the report.

The impact of the greedy algorithm initial solutions were rather minimal in terms of finding solutions more quickly. Having a better initial lower bound helps by allowing for more pruning earlier, making the process of finding a solution simpler since there are less nodes to process. In that way, the initial BSSF binding played a role helping increase solution count, but not as much as the priority queue. The impact was there; however, when I changed from a random solution to the greedy solution for the initial BSSF, the solution count and pruned states count increased immediately.

The bigger impacting mechanic was the priority key used in the heap queue implementation for storing the states. As mentioned above, I got a priority key using the current cost of the path divided by path length. This was done with the intention to give incentive to prioritize a path that is longer over a short path that may have a promising low initial bound. I found this approach to be quite effective in terms of getting solutions quickly and updating the BSSF more regularly.

For example, assume there were two states with a cost of 4000, one had a path of length 2 and one had a path of length 4. The path with length 2 would have a priority key of 2000, while the one of length 4 would have a priority key of 1000. In this case the heap queue is shaped with more emphasis placed on the length of the path, rather than the current cost since division is a very strong way to reduce a number and mechanically rewards the longer paths with higher priority.

Again, referring to the assumed case, if the 2-city path would need a cost of 2000 in order to tie in terms of priority. (The Heap Queue breaks ties to the left, not for any specific reason, just convenience). If this were the case, the shorter path would still go to the top of the queue since the current cost is just so low it overcomes the higher factor of division of the longer path. If that were the case, it wouldn’t be a bad idea to prioritize the node since the extremely low cost makes it’s a promising candidate for a good solution.

In retrospect, while I think my approach was not a bad way to handle taking path length into consideration, it is no where near a perfect solution. Even if the heap queue takes the longest path and places, it at the top since it is near a solution, that in no way guarantees that all remaining paths will be valid. The path may be near the end but could still be a dud and not produce any solutions if all the remaining paths are non-existing. I think there could be a balance struck between using states with a longer path, states with a low cost, and states that seem promising based on cost matrix.

The last factor (how to judge a state as promising) especially interests me now that I finished this project. Maybe there is a good way to look at the ratio of remaining paths of length infinity for cities not visited yet and make that factor into the calculation of a priority key. That way you could greatly penalize states that may be close to finishing but will likely end with no solution and avoid wasting time on ones that won’t work. I’m sure there are many approaches out there that have considered ideas like this and I am excited to take a look at some of them for Project 6 now that some ideas are starting to come to me since I’ve finished my implementation for this project.

# Source Code Files

TSPSolver.py

#!/usr/bin/python3  
import copy  
  
from which\_pyqt import PYQT\_VER  
  
if PYQT\_VER == 'PYQT5':  
 from PyQt5.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  
  
# Binary Heap implementation of a Priority Queue. The code is essentially the same from my project 3, with a few  
# specializations to make it work for this project (using a unique priority key instead of values from dist array and  
# other changes. These changes don't affect the performance of this data structure. The time complexity of  
# each function is detailed before, with insert, decrease\_key, and delete\_min all performing at a time complexity  
# of O(log n) because they need to sort the values up or down depending on the function, which takes about log n  
# iterations to complete on average or worst case.  
#  
# Unlike the previous project, the space complexity of the heap queue is more interesting than before. Because of the  
# way the branch and bound function trims and prunes states, you do not necessarily get a heap queue with O(2^n)  
# exponential side because not every state is added to the queue. It is entirely dependent on the problem to  
# solve, but when testing my code, a TSP problem with 15 or less cities rarely had a max queue size of  
# more than 100. For a 10 city problem, 100 max queue size would be a size complexity of O(n^2), so that is a good  
# starting evaluation of space complexity for the heap queue in this context. There were always outliers in testing  
# for max queue size, but in general, it was very rare to see a max queue size greater than O(n^2) where n is the  
# number of cities to visit, making O(n^2) a safe upper bound on space complexity.  
class HeapQueue:  
 def \_\_init\_\_(self):  
 # Initialize empty array of nodes.  
 self.nodes = []  
  
 # Using a dictionary for the array indices to match Node object to location in the heap.  
 self.array\_indices = {}  
  
 # Max queue size variable used to track the largest queue size each time a node is added.  
 self.max\_queue\_size = 0  
  
 # The delete\_min function is essential the same from before for Project 3 with the main change being to the way  
 # priority is calculated. Before, the queue was used to sort the nodes by the lowest distance. In this case, the  
 # priority key is decided from the result of (node.lowerbound / len(node.path)). Initially I had just used the  
 # lower bound as the sorting mechanism for the heap queue, and saw that in testing, I was often getting a good  
 # answer rather quickly, but was not pruning many states since the bssf was not updating regularly. To fix this,  
 # I added the part about dividing by the length of the current path. The thought behind this was to help push states  
 # that may have a slightly higher lower bound above shorter paths with a smaller lowerbound but less cities  
 # visited so far. The result was more states being pruned and the bssf being updated more regularly as a result,  
 # so I think it accomplished my goal in terms of improvement from just using the lowerbound for the key.  
 #  
 # The time complexity for this function is O(log n), since as the heap grows bigger, it grows exponentially in size  
 # but gets deeper at log(n) rate. Since there are log(n) levels (ex, if there are 64 nodes in the heap, there are  
 # log base 2 of 64 = 6 levels of the heap queue). At worst only log(n) traversals can be made, so the time  
 # complexity is O(log n).  
 def delete\_min(self):  
  
 # Parent index is 0  
 parent\_index = 0 # Initialize variable - Time O(1)  
  
 # Save node at top of heap  
 node\_to\_remove = self.nodes[parent\_index] # Get value from array - Time O(1)  
  
 # Get node at end of heap  
 heap\_end\_node = self.nodes[self.size() - 1] # Get value from array - Time O(1)  
  
 # Remove node from end of heap  
 self.nodes.remove(heap\_end\_node) # Remove value from end of array - Time O(1)  
  
 # If node at top was final node,  
 if self.size() == 0: # Check if condition  
 del self.array\_indices[node\_to\_remove] # Delete key, value pair from dictionary - Time O(1)  
 return node\_to\_remove  
  
 # Set top node to final node in heap  
 self.nodes[parent\_index] = heap\_end\_node # Update value in array - Time O(1)  
  
 # Set removed node index to -1  
 del self.array\_indices[node\_to\_remove] # Delete key, value pair from dictionary - Time O(1)  
  
 # Set end nodes index to 0/parent node since it is at top now  
 self.array\_indices[heap\_end\_node] = parent\_index # Update value in dictionary for key - Time O(1)  
  
 # Continue to loop until node shifted to top of heap is in correct location  
 # Time O(log n), since half the tree is reduced each time the node moves and has log n depth to traverse.  
 while True:  
  
 # Get Index of left and right children  
 left\_child\_index = (parent\_index \* 2) + 1 # Calculate index - O(1)  
 right\_child\_index = (parent\_index \* 2) + 2 # Calculate index - O(1)  
  
 # Get parent node and lowerbound  
 parent\_node = self.nodes[parent\_index] # Get value from array - Time O(1)  
 parent\_lower\_bound = parent\_node.lowerbound # Save value from Node object - Time O(1)  
  
 # Get custom priority key using lower bound divided by length of the current path.  
 parent\_priority = parent\_lower\_bound / len(parent\_node.path) # Calculate value and save - Time O(1)  
  
 # If left child exists in array  
 if left\_child\_index < self.size():  
 left\_child\_node = self.nodes[left\_child\_index] # Get value from array - Time O(1)  
 left\_child\_lower\_bound = left\_child\_node.lowerbound # Save value from Node object - Time O(1)  
 left\_child\_priority = left\_child\_lower\_bound / len(left\_child\_node.path) # Calculate value - Time O(1)  
 # If there is no left child (and by extension no right child)  
 else:  
 left\_child\_priority = np.inf # Set value - Time O(1)  
 left\_child\_node = None # Set value - Time O(1)  
  
 # If right child exists in array  
 if right\_child\_index < self.size():  
 right\_child\_node = self.nodes[right\_child\_index] # Get value from array - Time O(1)  
 right\_child\_lower\_bound = right\_child\_node.lowerbound # Save value from Node object - Time O(1)  
 right\_child\_priority = right\_child\_lower\_bound / len(right\_child\_node.path) # Calculate value-Time O(1)  
 else:  
 right\_child\_priority = np.inf # Set value - Time O(1)  
 right\_child\_node = None # Set value - Time O(1)  
  
 # Initialize value to update if switch happens  
 parent\_swapped = False # Variable initialization - Time O(1)  
  
 # If left child lowerbound is less than right, use it, tie break left  
 if left\_child\_priority <= right\_child\_priority: # Check if condition - Time O(1)  
 # If left child is less than parent, update, else do nothing  
 if left\_child\_priority < parent\_priority: # Check if condition - Time O(1)  
 self.nodes[parent\_index] = left\_child\_node # Update value in array - Time O(1)  
 self.nodes[left\_child\_index] = parent\_node # Update value in array - Time O(1)  
  
 self.array\_indices[parent\_node] = left\_child\_index # Update value in array - Time O(1)  
 self.array\_indices[left\_child\_node] = parent\_index # Update value in array - Time O(1)  
  
 parent\_index = left\_child\_index # Update variable value - Time O(1)  
 parent\_swapped = True # Update variable value - Time O(1)  
 else: # If right child lowerbound is less than parent, check for swap  
 if right\_child\_priority < parent\_priority: # Check if condition - Time O(1)  
 self.nodes[parent\_index] = right\_child\_node # Update value in array - Time O(1)  
 self.nodes[right\_child\_index] = parent\_node # Update value in array - Time O(1)  
  
 self.array\_indices[parent\_node] = right\_child\_index # Update value in array - Time O(1)  
 self.array\_indices[right\_child\_node] = parent\_index # Update value in array - Time O(1)  
  
 parent\_index = right\_child\_index # Update variable value - Time O(1)  
 parent\_swapped = True # Update variable value - Time O(1)  
  
 # If heap did not change on this iteration, break out of while loop and return.  
 if not parent\_swapped: # Check if condition - Time O(1)  
 break  
  
 # Remove node removed from top of heap queue  
 return node\_to\_remove # Return value - Time O(1)  
  
 # The decrease\_key also works the same as from Project 3 with the exception of the priority key. decrease\_key, like  
 # delete\_min, uses the (node.lowerbound / len(node.path)) formula to find a balance between pushing up nodes with  
 # a smaller bound and prioritizing nodes with a longer path in order to update the bssf sooner. Decrease\_key is  
 # called each time a node is inserted to the heap, moving it up from the last spot in the heap until the priority  
 # is greater than the parent node.  
 #  
 # Same with delete\_min, the time complexity of this function is O(log n), since as the heap grows, there are  
 # log n levels of depth that a node can travel up or down, making O(log n) an upper bound on the times the while  
 # loop of percolating the node.  
 def decrease\_key(self, node):  
 # Check if node has already been popped off of the queue then return early.  
 if node not in self.array\_indices: # Check if condition - Time O(1)  
 return  
  
 # Get index of child just added to heap queue  
 child\_index = self.array\_indices[node] # Save value from array - Time O(1)  
  
 # Percolate children up if parent needs to move  
 # Time is O(log n) since there are log n levels of depth to the heap, so a worst case/average case  
 # may iterate log n times for a larger heap.  
 while True:  
  
 # New node reached top of heap, stop percolating  
 if child\_index == 0: # Check if condition - Time O(1)  
 break  
  
 # Get node from array using index  
 child\_node = self.nodes[child\_index] # Save value from array - Time O(1)  
  
 # Get parent index using child index - 1 // 2  
 parent\_index = (child\_index - 1) // 2 # Calculate value - Time O(1)  
 parent\_node = self.nodes[parent\_index] # Save value from array - Time O(1)  
  
 # Create custom priority key using lower bound divided by length of the current path.  
 parent\_priority = parent\_node.lowerbound / len(parent\_node.path) # Calculate value - Time O(1)  
  
 child\_priority = child\_node.lowerbound / len(child\_node.path) # Calculate value - Time O(1)  
  
 # If the child has a lower priority key than the parent, shift it upwards in the heap.  
 if child\_priority < parent\_priority: # Check if condition - Time O(1)  
  
 # Switch nodes in the tree for parent and child.  
 self.nodes[child\_index] = parent\_node # Update value in array - Time O(1)  
 self.nodes[parent\_index] = child\_node # Update value in array - Time O(1)  
  
 # Update index of each node in array indices dictionary.  
 self.array\_indices[child\_node] = parent\_index # Update value in dictionary - Time O(1)  
 self.array\_indices[parent\_node] = child\_index # Update value in dictionary - Time O(1)  
  
 # Update variable for child index for next iteration.  
 child\_index = parent\_index # Variable assignment - Time O(1)  
  
 else: # If the child priority is greater than parent, no change is made and the loop is broken.  
 break  
  
 # Function to insert the node to the end of the heap queue, then move it to the right spot by calling decrease\_key.  
 # Nothing changed from before for this function, except for the addition of a check to update the max\_queue\_size  
 # variable in order to keep track of the largest size of the heap as states are added, removed, and pruned.  
 #  
 # The actual inserting of the node is O(1) since it is just adding to the end of the array, but the call to  
 # decrease\_key makes the time complexity of this function O(log n), since it needs to be shifted to the right  
 # location in the tree.  
 def insert(self, node):  
 # Add node to array storage  
 self.nodes.append(node) # Append value to array - Time O(1)  
  
 # Check if queue has new max size after appending new node, update max queue if true.  
 if self.size() > self.max\_queue\_size: # Check if condition - Time O(1)  
 self.max\_queue\_size = self.size() # Variable assignment - Time O(1)  
  
 # Save array index to indices dictionary using node as key  
 self.array\_indices[node] = len(self.nodes) - 1 # Save value to dictionary - Time O(1)  
  
 # Check for shifting node up  
 self.decrease\_key(node) # Call to decrease\_key - Time O(log n)  
  
 # Return number of nodes in heap queue, used occasionally in the heap functions to get location of end node,  
 # and other tasks.  
 def size(self):  
 return len(self.nodes) # Return value from len - Time O(1)  
  
  
# Node is the class used to represent search states in this program. A search state Node has 3 main arrays; the   
# cost matrix array that contains the current values for each possible path as they are updated, the path array which   
# contains the actual city objects of each city that has been visited that is used by TSPSolution to actually generate  
# a solution object, and the cities\_visited array which stores the index of the cities that have been visited so far.   
# The cities\_visited array is not needed for creating the end solution, but since the cost matrix operates in terms of  
# source/destination path pairs, it was a lot easier to the index of each city ready to go instead of getting them from  
# the cities array every time, even at the cost of additional space to store the indices.   
#  
# Inside of the node class, there are functions to calculate the cost matrix for a city visited, as well as to check   
# and see if a path is a valid solution. The node.lowerbound stores the bound/cost of the current path, and is   
# referenced often through the class and branch and bound solution to check for pruning and binding of states.  
#  
# Time and space complexity of each individual function is given in the comments above each one. In general however,  
# functions that solve/reduce the cost matrix are of time complexity O(n^2) since they are iterating over the complete  
# n x n array. The overall space complexity of each node is dictated by the same n x n or O(n^2) size array that   
# is used for the cost matrix.  
class Node:  
 # Init function for Node class. For the initial state, most of the values are set to None or zero. However, for  
 # each child node, the node is initialized using the cost matrix, path, and cities visited array of the parent  
 # node. The cost matrix is copied using deepcopy, which results in the overall time complexity of O(n^2), but  
 # simplifies the copying process instead of using nested loops. The other arrays are copied using loops instead  
 # of deep copy since deep copy was slowing the code down significantly (30 to 60 seconds in the worst cases).  
 #  
 # Time complexity - O(n^2) for deep copying the n x n cost matrix.  
 # Space complexity - O(n^2) in creating the n x n cost matrix using deep copy. Space is also used to create the  
 # path and cities visited arrays, but they are only n length 1d arrays, so the overall space complexity doesn't  
 # change.  
 def \_\_init\_\_(self, lower\_bound, parent\_matrix, curr\_city, curr\_city\_index, parent\_path, parent\_cities):  
 # Deep copy cost matrix passed in from parent node.  
 self.costmatrix = copy.deepcopy(parent\_matrix) # Deep copy - Time O(n^2)  
  
 # Assign new lower bound to lower bound from parent.  
 self.lowerbound = lower\_bound  
  
 # Initialize empty path array to save parent cities to  
 self.path = [] # Initialize empty array - Time O(1)  
  
 # Copy cities in path from parent - Time O(n)  
 for city in parent\_path:  
 # Append value to array - Time O(1)  
 self.path.append(city)  
  
 # Append current city this state represents to the path array - Time O(1)  
 self.path.append(curr\_city)  
  
 # Initialize empty cities visited array to save cities visited from parent node to.  
 self.cities\_visited = [] # Initialize empty array - Time O(1)  
  
 # Copy index of cities visited in path from parent - Time O(n)  
 for city in parent\_cities:  
 # Append value to array - Time O(1)  
 self.cities\_visited.append(city)  
  
 # Append current index of the city this state represents to the path array - Time O(1)  
 self.cities\_visited.append(curr\_city\_index)  
  
 # A function that is only used once by the initial state to generate the numpy array used as the cost matrix.  
 # This function is not called when creating children states, a deep copy is made by the init function to pass on  
 # a new array object in the same current state. This function builds the array from the list of cities passed on  
 # from the GUI. The cities are added in the order they are found in the cities list. After creating the array,  
 # the cost of each path to and from each city is added to the array at the indices of each source/destination  
 # path combination.  
 #  
 # The array generation itself is an O(n^2) time complexity since it initializes the array of that size with zeroes  
 # for each value. Aside from just the array generation, the function then loops through each row and column of the  
 # array to check and update the value based on the scenario, which is also a time O(n^2) section of code, since the  
 # array has rows and columns equal to the number of cities to visit.  
 #  
 # Similar to time complexity, the array generated is an n x n array where both length and width are the number  
 # of cities to visit, making the space complexity of this function O(n^2)  
 def generateCostMatrix(self, cities):  
 # Initialize n x n cost matrix using np.zeroes.  
 # Time complexity - Time O(n^2) to initialize each values.  
 # Space complexity - O(n^2) for n x n array.  
 self.costmatrix = np.zeros((len(cities), len(cities)))  
  
 # Loop through each row and column and get the correct value for each path to and from each city.  
 # Time complexity O(n^2) - Nested for loop for each row and column of n x n array.  
 for sourceindex, city in enumerate(cities):  
 for destindex, secondcity in enumerate(cities):  
  
 # If current path checking is to and from same city, set to infinity since there is no path.  
 if city == secondcity: # Check if condition - Time O(1)  
 self.costmatrix[sourceindex][destindex] = np.inf # Update values in array - Time O(1)  
 else:  
 # If path exists, get cost using costTo function and save to array.  
 self.costmatrix[sourceindex][destindex] = city.costTo(secondcity) # Update value - Time O(1)  
  
 # reduceMatrix is nearly identical in function to the get\_initial\_lower\_bound function below. Both of them  
 # center around the behavior of getting the minimum values for the rows and columns in order to reduce the cost  
 # matrix for a child state. However, the child states also need to account for the current path being taken as  
 # well as account for the cost of the parent path up to this point, making it easier to have the two functions be  
 # separate.  
 #  
 # Although there are a couple additional parts to this function, there is no difference in overall time complexity  
 # compared to get\_initial\_lower\_bound, since the nested for loops still dictate the time complexity of this  
 # function. Both times, every value is iterated over and updated if needed, making time complexity here O(n^2)  
 def reduceMatrix(self, cities, dest\_city\_index):  
 # Get previous city index from cities\_visited array  
 source\_city\_index = self.cities\_visited[len(self.cities\_visited) - 2] # Get value from array - Time O(1)  
  
 # Get cost of selected path from cost matrix  
 path\_cost = self.costmatrix[source\_city\_index][dest\_city\_index] # Get value from array - Time O(1)  
  
 # Set path and inverse to infinity in cost matrix  
 self.costmatrix[source\_city\_index][dest\_city\_index] = np.inf # Assign value in array - Time O(1)  
 self.costmatrix[dest\_city\_index][source\_city\_index] = np.inf # Assign value in array - Time O(1)  
  
 # Set row of source city and column of dest city to inf  
 # We update one row and a column together in one for loop, so time is O(n) instead of O(n^2)  
 for city\_index, city in enumerate(cities):  
 self.costmatrix[source\_city\_index][city\_index] = np.inf # Update value in array - Time O(1)  
 self.costmatrix[city\_index][dest\_city\_index] = np.inf # Update value in array - Time O(1)  
  
 # Get the minimum value of each row using numpy.  
 rowminvalues = np.amin(self.costmatrix,  
 axis=1) # numpy.amin - Time O(n^2) (has to check every value for min)  
  
 # Loop over each row and reduce each value in each column of the row by the min value.  
 # Time O(n^2) since each row and column contains every city to visit.  
 for rowindex, city in enumerate(self.costmatrix):  
 # Get min value from min value by row array  
 reducevalue = rowminvalues[rowindex] # Get value from array - Time O(1)  
  
 # If min value is infinity then set to 0 in min value array to get correct updated lower bound  
 if reducevalue == np.inf: # Check if condition - Time O(1)  
 rowminvalues[rowindex] = 0 # Assign value in array - Time O(1)  
  
 for colindex, secondcity in enumerate(self.costmatrix):  
 # If value to reduce is 0 or infinity, don't update values in row  
 if self.costmatrix[rowindex][colindex] != 0 and reducevalue != np.inf: # Check if condition - Time O(1)  
 # Update value in current column  
 self.costmatrix[rowindex][colindex] -= reducevalue # Update variable - Time O(1)  
  
 # Get minimum values for each column using numpy.amin  
 colminvalues = np.amin(self.costmatrix, axis=0) # numpy.amin() - Time O(n^2)  
  
 # Loop over each column and reduce each value in each row of the column by the min value.  
 # Time O(n^2) since each row and column contains every city to visit.  
 for colindex, city in enumerate(self.costmatrix):  
 # Get min value from min value by column array  
 reducevalue = colminvalues[colindex] # Get value from array - Time O(1)  
  
 # If min value is infinity then set to 0 in min value array to get correct updated lower bound  
 if reducevalue == np.inf: # Check if condition - Time O(1)  
 colminvalues[colindex] = 0 # Assign value in array - Time O(1)  
  
 for rowindex, secondcity in enumerate(self.costmatrix):  
 # If value to reduce is 0 or infinity, don't update values in column  
 if self.costmatrix[rowindex][colindex] != 0 and reducevalue != np.inf: # Check if condition - Time O(1)  
 # Update value in current row  
 self.costmatrix[rowindex][colindex] -= reducevalue # Update variable - Time O(1)  
  
 # Get the sum of the minimum values used to reduce the matrix  
 reduction\_cost = np.sum(rowminvalues) + np.sum(colminvalues) # np.sum for both arrays, Time O(n^2)  
  
 # Get new bound by adding previous lower bound, path cost and reduction cost  
 self.lowerbound += reduction\_cost + path\_cost # Update variable - Time O(1)  
  
 # The initial function used by the first state created in the branch and bound method to get the starting  
 # lower bound value for the rest of the children state to use in their calculations. Functions is extremely  
 # similar to reduceMatrix function above, and likely could be combined, but because the cost of the lowerbound  
 # of the children nodes rely on the initial lowerbound, the calculations are slightly different, so I ended up  
 # having the two functions split since they were developed that way.  
 #  
 # The cose uses numpy.amin to extract the minimun values from each row or column, then iterates over the  
 # entire array to update the values as needed (if they are not infinity or 0). The rows are handled first, since  
 # updating the rows may change if there is a need to reduce the column values. For either rows or columns, every  
 # value has to be checked and updated, so each nested for loop has a time complexity of O(n^2), since the cost  
 # matrix itself has a space complexity of O(n^2), where n is the number of cities to try and visit. Even though  
 # there are two sets of nested for loops, the overall time complexity of this function is O(n^2), since every  
 # value of the array has to be checked.  
 def get\_initial\_lower\_bound(self):  
 # Get the minimum value of each row using numpy.  
 rowminvalues = np.amin(self.costmatrix, axis=1) # numpy.amin - Time O(n^2) (has to check every value for min)  
  
 # Loop over each row and reduce each value in each column of the row by the min value.  
 # Time O(n^2) since each row and column contains every city to visit.  
 for rowindex, city in enumerate(self.costmatrix):  
 for colindex, secondcity in enumerate(self.costmatrix):  
 # Extract minimun value for this row from rowminvalues array - Time O(1)  
 reducevalue = rowminvalues[rowindex]  
  
 # If value to reduce is 0 or infinity, don't update values in row  
 if self.costmatrix[rowindex][colindex] != 0 and reducevalue != np.inf: # Check if condition - Time O(1)  
 # Update value in current column  
 self.costmatrix[rowindex][colindex] -= reducevalue # Variable assignment - Time O(1)  
  
 # Get minimum values for each column using numpy.amin  
 colminvalues = np.amin(self.costmatrix, axis=0) # numpy.amin() - Time O(n^2)  
  
 # Loop over each column and reduce each value in each row of the column by the min value.  
 # Time O(n^2) since each row and column contains every city to visit.  
 for colindex, city in enumerate(self.costmatrix):  
 for rowindex, secondcity in enumerate(self.costmatrix):  
 # Extract minimun value for this column from colminvalues array - Time O(1)  
 reducevalue = colminvalues[colindex]  
  
 # If value to reduce is 0 or infinity, don't update values in column  
 if self.costmatrix[rowindex][colindex] != 0 and reducevalue != np.inf: # Check if condition - Time O(1)  
 # Update value in current column  
 self.costmatrix[rowindex][colindex] -= reducevalue # Variable assignment - Time O(1)  
  
 # Calculate the initial lower bound from the sum of the minimun values used to reduce the rows and columns  
 self.lowerbound += np.sum(rowminvalues) + np.sum(colminvalues) # Sum rows and columns - Time O(n^2)  
  
 # The main function used by the greedy algorithm to calculate what is the shortest path from the current  
 # city to any other city in the array. It is a greedy function, because it simply chooses the best path at  
 # the time, even if the final solution after that choice may not exist. It is only concerned with getting the  
 # lowest cost path from this state.  
 #  
 # Every time this function is called, it iterates over the list of cities. It does skip over iterations where  
 # the city has already been visited, but that does not change the fact that the for loop still iterates for that  
 # city to start. This for loop over the array of cities makes the time complexity of this function O(n), where n  
 # is the number of cities to visit.  
 def get\_greedy\_path(self, cities):  
 # Get the index of the previous city visited  
 source\_index = self.cities\_visited[len(self.cities\_visited) - 1] # Get value from array - Time O(1)  
  
 # Get values of paths to each city for current source city from cost matrix.  
 # Slicing row out of array - Time O(n), where n = number of cities to visit (length of array sliced)  
 # Space complexity of this array is O(n), as it is taking one row out of the n by n cost matrix array.  
 city\_row = self.costmatrix[source\_index, :]  
  
 # Initialize values to save minimum cost and index to  
 dest\_index = -1 # Variable initialization - Time O(1)  
 min\_cost = np.inf # Variable initialization - Time O(1)  
  
 # For each city in list, check for shortest path and save cost and array. Time O(n)  
 for index, city in enumerate(city\_row):  
 # Skip checking for any cities not visited yet  
 if index in self.cities\_visited: # Check if condition - Time O(1)  
 continue  
  
 # Check for min value  
 if city\_row[index] < min\_cost: # Check if condition - Time O(1)  
 # Update minimum cost and index values.  
 min\_cost = city\_row[index] # Variable assignment - Time O(1)  
 dest\_index = index # Variable assignment - Time O(1)  
  
 # Check if minimum cost is infinity - If so, there is no valid solution for this path.  
 if min\_cost == np.inf: # Check if condition - Time O(1)  
 self.lowerbound = min\_cost # Variable assignment - Time O(1)  
  
 # If minimum cost is not infinity, the current path is still okay, update cost and path arrays  
 else:  
 # Append index of new city visited to array - Time O(1)  
 self.cities\_visited.append(dest\_index)  
  
 # Append city object from cities list to array - Time O(1)  
 self.path.append(cities[dest\_index])  
  
 # Update current total cost of path saved in lowerbound - Time O(1)  
 self.lowerbound += min\_cost  
  
 # Check if current path is complete, if so, check for valid path back to starting city.  
 if self.test(cities): # Check if condition - Time O(1)  
  
 # Get cost of path from final city to starting city  
 return\_path = self.costmatrix[dest\_index][self.cities\_visited[0]] # Get value from array - Time O(1)  
  
 # Path to starting node equals infinity, there is no valid return path, don't update solution  
 self.lowerbound += return\_path # Variable assignment - Time O(1)  
  
 # A simple function that returns if a path is complete or not. A path is complete if every city is  
 # in the array, checked for using the length of the cities array and the path array containing  
 # TSPClasses.City objects.  
 #  
 # It is only a simple len comparison, so the function is O(1) time, no matter what the result is.  
 def test(self, cities):  
 # Incomplete path, return infinity  
 if len(self.path) != len(cities): # Check if condition - Time O(1)  
 return False # Return false - O(1)  
  
 # If path is complete, return lower bound  
 return True # Return true - O(1)  
  
 # Debugging function used to print cost matrix with indices and tab spacing for easy viewing. This is not used  
 # any where in the actual solution, so I did not comment it more specifically, but I did not remove it so it  
 # can be used for Project 6 when implementing the final method.  
 def printCostMatrix(self):  
 arrayString = "X\t"  
 val = 0  
 while val < len(self.costmatrix):  
 arrayString += str(val) + "\t\t\t"  
 val += 1  
  
 arrayString += '\n'  
  
 for sourceindex, city in enumerate(self.costmatrix):  
 arrayString += str(sourceindex) + "\t"  
 for destindex, secondcity in enumerate(self.costmatrix):  
 if self.costmatrix[sourceindex][destindex] == np.inf or self.costmatrix[sourceindex][destindex] == 0:  
 arrayString += str(self.costmatrix[sourceindex][destindex]) + "\t\t\t"  
 else:  
 arrayString += str(self.costmatrix[sourceindex][destindex]) + "\t\t"  
  
 arrayString += '\n'  
  
 print(arrayString)  
  
  
class TSPSolver:  
 def \_\_init\_\_(self, gui\_view):  
 self.\_scenario = None  
  
 def setupWithScenario(self, scenario):  
 self.\_scenario = scenario  
  
 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  
  
 # A greedy algorithm implementation of the TSP solver. While intended for Project 6, I decided to make it earlier  
 # to use it to get an initial bssf state for the branch and bound function, since it would get me a decent estimate  
 # resulting in more states pruned earlier and less states to create as a result.  
 #  
 # The function works as described in the specs; it starts from each city and checks the values for all possible  
 # paths from the current city to any city not visited and selects the smallest one, adding it to the current path.  
 # If it ever encounters a minimum value of infinity, the path is deemed impossible and no solution is added.  
 # The function tries this greedy approach for every starting city, updating the best solution from each of the  
 # starting points, returning what ever is the best solution.  
 #  
 # The greedy approach has a time complexity of O(n^3) since there are essentially three nested loops going on.  
 # The for loop that determines the starting city iterates over the length of the cities array, giving us O(n) time  
 # for the outer-most loop. Inside, there is a while loop that calls the get\_greedy\_path function until the path  
 # is complete (len(path) == len(cities)). While this while loop will break early if the algorithm encounters an  
 # invalid solution, on average it will run through the length of the cities array to get a solution, giving us  
 # another O(n) time loop. Lastly, the call to get\_greedy\_path is the third nested loop, since the function itself  
 # loops over the length of the cities array to check for the minimum value of the paths out of the current city,  
 # resulting in the third O(n) time complexity loop. For a perfect array that has all valid solutions for each  
 # starting city, that would give us O(n^3) time complexity as an upper bound for time complexity.  
 #  
 # Space complexity for the greedy solution is simply O(n^2), since we are not creating many states like in the  
 # branch and bound method, just one that has a n x n array for the cost matrix that is referenced the entire time.  
 def greedy(self, time\_allowance=60.0):  
 # Initialize dictionary for results  
 results = {} # Initialize variable - Time O(1)  
  
 # Get list of cities  
 cities = self.\_scenario.getCities() # O(n) time to get array of cities  
  
 # Save number of cities to variable for easy use.  
 ncities = len(cities) # Get len - Time O(1)  
  
 # Initialize solution count to count number of valid solutions found  
 solution\_count = 0 # Initialize variable - Time O(1)  
  
 # Initialize solution to infinity to avoid None error  
 greedy\_solution = None # Initialize variable - Time O(1)  
  
 # Start timer to check if time allowed is expired.  
 start\_time = time.time() # Get time - Time O(1)  
  
 # Try each city as a starting spot for the greedy path algorithm. Try each city.  
 # Each city is looped over - Time O(n)  
 for starting\_city\_index in range(ncities):  
  
 # Check if time expired, if so, break and return best solution found so far.  
 if time.time() - start\_time > time\_allowance: # Check if condition - Time O(1)  
 break  
  
 # Initialize starting state for path starting from indicated city.  
 # Node.\_\_init\_\_ - Time O(1) since there are no parent arrays to copy  
 initial\_state = Node(0, [], cities[starting\_city\_index], starting\_city\_index, [], [])  
  
 # Generate cost matrix for this state from parent array.  
 initial\_state.generateCostMatrix(cities) # Time - O(n^2)  
  
 # Loop until greedy solution finds answer, will loop over all cities if there is a valid solution.  
 # Time - O(n) since the while loop will break once every city has been visited.  
 while not initial\_state.test(cities):  
  
 # Call function get\_greedy\_path to get min path from current city and add to path.  
 initial\_state.get\_greedy\_path(cities) # Time O(n)  
  
 # There is no valid solution for this matrix  
 if initial\_state.lowerbound == np.inf: # Check if statement - Time O(1)  
 break  
  
 # If there is no valid solution yet, add current solution if valid.  
 if greedy\_solution is None: # Check if statement - Time O(1)  
 # Check that initial solution is valid  
 if initial\_state.lowerbound != np.inf: # Check if statement - Time O(1)  
 # Found first valid solution, create TSPSolution and save for greedy\_solution.  
 greedy\_solution = TSPSolution(initial\_state.path) # Create solution - Time O(n)  
  
 # Increment number of solutions found.  
 solution\_count += 1 # Increment variable - Time O(1)  
 else:  
 # Greedy solution already exists, check if the new solution is better than previous one.  
 if initial\_state.lowerbound < greedy\_solution.cost: # Check if statement - Time O(1)  
 # Found a better solution, create TSPSolution and save for greedy\_solution.  
 greedy\_solution = TSPSolution(initial\_state.path) # Create solution - Time O(n)  
  
 # Increment number of solutions found.  
 solution\_count += 1 # Increment variable - Time O(1)  
  
 # Stop timing for end timer  
 end\_time = time.time() # Get time - Time O(1)  
  
 # Save values to results array - Time O(1)  
 results['cost'] = greedy\_solution.cost if solution\_count > 0 else math.inf  
 results['time'] = end\_time - start\_time  
 results['count'] = solution\_count  
 results['soln'] = greedy\_solution  
 results['max'] = None  
 results['total'] = None  
 results['pruned'] = None  
  
 # Return result of greedy approach - Time O(1)  
 return results  
  
 # The function used by the branch and bound method to expand a node into child states for each possible path  
 # out of the current city. The current state is passed in and its values are used to create the new states since  
 # the node.\_\_init\_\_ function copies the arrays of the parent node. Expand iterates over the list of cities,  
 # checks that the current city is not already on the path of the node. If it is, the city is skipped. If the  
 # city has not been visited yet, then a new state is created using the city, which is added to the end of the path.  
 # The new state is then passed into the result array T, which is ultimately returned to the main branch and bound  
 # method.  
 #  
 # The time complexity of this function is O(n^2) because of the node creation process. The expand function itself  
 # is simply and only relies on a for loop to iterate over the list of cities and create nodes for cities that  
 # have not been visited, making it a O(n) function. However, the init function for creating a new node is O(n^2)  
 # since the n x n cost matrix has to be copied from the parent state to the new child. This is handled with  
 # copy.deepcopy(), which ensures there is not a just a copying of reference to the array, but adds time to the  
 # process. The call to reduceMatrix is also O(n^2), ensuring the time complexity of the function to be O(n^2), not  
 # O(n)  
 #  
 # The space complexity of this function is more complicated. Each node created is of size O(n^2) since it contains  
 # a unique n x n array for the cost matrix of this unique state. As the states get deeper, less nodes are created  
 # per call to expand, but as many as n-1 nodes can be created when expand is called. These nodes are stored in  
 # a result array, T, which has n-1 size at its biggest. This makes the space complexity of this function to be  
 # O(n^3), since almost n nodes of size n^2 can be created in one call.  
 def expand(self, curr\_state, cities):  
 # Create an empty array to hold all the state nodes created.  
 T = [] # Initialize empty array - Time O(1)  
  
 # Iterate over all cities in the list - Time O(n)  
 for index, city in enumerate(cities):  
  
 # Only create nodes for cities that have not been visited yet.  
 if index not in curr\_state.cities\_visited: # Check if statement - Time O(1)  
 # Create new state  
 new\_state = Node(curr\_state.lowerbound, curr\_state.costmatrix, city, index, curr\_state.path,  
 curr\_state.cities\_visited) # Time O(n^2) to copy arrays.  
  
 # Reduce matrix of new state  
 new\_state.reduceMatrix(cities, index) # Time O(n^2) - time to reduce cost matrix.  
  
 # Append to result array  
 T.append(new\_state) # Append to end of array - Time O(1)  
  
 # Return array of state nodes.  
 return T # Return value - Time O(1)  
  
 # This is the function for the core branch and bound algorithm. The initial state is created from the array of  
 # cities from the scenario given by the GUI. The branch and bound solution is always started from city at index 0.  
 # After the initial state's cost matrix and path is handled and the starting lower bound is found, the heap queue  
 # is created and an initial bssf using the greedy algorithm is found. From there, the looping begins;  
 # for each iteration, the state with the lowest priority key is popped from the heap and expanded to child states.  
 # Each of the child states is check for a valid solution. If a solution is found, then the cost is compared to the  
 # bssf. If the cost is an improvement, the solution is added. If the current state is not a complete solution, it  
 # is checked against the bound for pruning. If the current state is under the bound, it is pushed onto the heap,  
 # else nothing happens and the state is pruned. Each iteration of the while loop checks the elapsed time, breaking  
 # the loop at its head if the expected time allowance is passed.  
 #  
 # The time complexity of the function is still the exponential time of O(2^n). The branching and bounding solution  
 # ensures that we get closer to a solution by not spending as much time working on states that do not go anywhere.  
 # That does not change the fact that we create all of the states using the expand function in order to check them  
 # for pruning. Each state results in an exponentially growing number of children states until a complete solution  
 # is found. The pruning reduces the number of states created, especially when a state exceeds the bound early into  
 # the path. The greedy algorithm returns a fairly lower bssf, which results in slightly higher early pruning  
 # than compared to when I ran branch and bound with the random solution. Still, as the total number of states  
 # created in my testing indicates, the number of states created is still an exponential number, making the while  
 # loop continue for an exponential time. Of course in our code, we use a time limit to stop the program after  
 # a max of one minute, but if the code were to continue, the run time would take a while since it has to loop  
 # until there are no more states left on the queue to loop through. Since every state create is a loop on the for  
 # loop, the time complexity is O(2^n)  
 #  
 # The space complexity for branch and bound is a bit more complicated. While the time is exponential, the space  
 # complexity of the function is not because of the pruning used. As mentioned in the comment above the heap queue,  
 # in testing, it was rare to see a heap queue max size result greater than O(n^2), with n being the number of  
 # cities to visit. Since the nodes contain the n x n cost matrix and each have a space of O(n^2), then we can  
 # say the heap queue has a space complexity of O(n^4), assuming the upper bound of the queue size is O(n^2).  
 # The nodes are still created by the expand function, which is as mentioned in the comments there,  
 # space complexity of O(n^3) since the nodes have a n x n array, and up to n - 1 nodes are created. If we consider  
 # the heap queue to have a space complexity upper bound of O(n^2) based on the max queue size results, and the  
 # expand function to return an array T of size O(n) containing nodes of size O(n^2), then our space complexity  
 # of the branch and bound function would be O(n^4), since the O(n^4) size of the heap queue with nodes at max size  
 # would dominate the O(n^3) size of the T array containing nodes created by expand.  
 def branchAndBound(self, time\_allowance=60.0):  
 # Create empty results array  
 results = {} # Create empty array - Time O(1)  
  
 # Get list of cities from scenario  
 cities = self.\_scenario.getCities() # Get cities - Time O(1)  
  
 # Initialize the initial state starting from city A  
 initial\_state = Node(0, [], cities[0], 0, [], []) # Time O(1) since there are no arrays to copy  
  
 # Create initial cost matrix for initial state  
 initial\_state.generateCostMatrix(cities) # Create initial matrix - Time O(n^2)  
  
 # Reduce and calculate initial lower bound  
 initial\_state.get\_initial\_lower\_bound() # Reduce initial matrix - Time O(n^2)  
  
 # Create empty Heap Queue and insert initial state  
 S = HeapQueue() # Create Heap Queue - Time O(1)  
 S.insert(initial\_state) # Insert initial state to heap queue - Time O(1) since there is nothing in the queue  
  
 # Initialize variables used in results array)  
 solution\_count = 0 # Initialize variable - Time O(1)  
 states\_pruned = 0 # Initialize variable - Time O(1)  
 total\_states = 1 # Initialize variable - Time O(1)  
  
 # Get starting bssf using greedy algorithm.  
 greedy\_result = self.greedy() # Greedy algorithm solution - Time O(n^3)  
 bssf = greedy\_result['soln'] # Save value from dictionary - Time O(1)  
  
 # Initialize timer to check for time out  
 start\_time = time.time() # Get time - Time O(1)  
  
 # While Heap Queue is not empty and time allowed has not passed - Time Complexity (See function analysis above)  
 while S.size() > 0 and time.time() - start\_time < time\_allowance:  
  
 # Get node at top of heap  
 P = S.delete\_min() # Delete min from Heap Queue - Time O(log n)  
  
 # Expand P into sub states, add to T  
 T = [] # Create empty list - Time O(1)  
 T = self.expand(P, cities) # Expand function - Time O(n^2)  
  
 # Update number of total states created by the expand function  
 total\_states += len(T) # Variable Assignment - Time O(1)  
  
 # For each state in T, check if is a solution or if it should be added to heap queue  
 for Pi in T: # Expand creates at most n - 1 nodes, so for loop is Time O(n)  
  
 # Reached a solution, compare cost to bssf  
 if Pi.test(cities): # Node.test - Time O(1)  
  
 # If the current solution has a lower cost than bssf, replace bssf  
 if Pi.lowerbound < bssf.cost: # Check if condition - Time O(1)  
 bssf = TSPSolution(Pi.path) # Create solution - Time O(n)  
  
 # Increment solution count since bssf was updated  
 solution\_count += 1 # Variable assignment - Time O(1)  
  
 # Not at a solution yet, check if state should be added to queue or pruned  
 else:  
 # Lower bound of partial solution is >= BSSF so prune the current state.  
 if Pi.lowerbound >= bssf.cost: # Check if condition - Time O(1)  
  
 # Update count of states pruned  
 states\_pruned += 1 # Variable assignment - Time O(1)  
  
 # Path lower bound is < BSSF push to heap queue and wait for next round  
 else:  
 S.insert(Pi) # Insert node to heap queue - Time O(log n)  
  
 # Stop timing for end timer  
 end\_time = time.time() # Get time - Time O(1)  
  
 # Push results of bssf and other variables to results array - Time O(1)  
 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'] = S.max\_queue\_size  
 results['total'] = total\_states  
 results['pruned'] = states\_pruned  
  
 # Return results - Time O(1)  
 return results  
  
 ''' <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

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()  
 #print( [c.\_index for c in listOfCities] )  
  
 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 ):  
 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:  
 #if difficulty == "Hard":  
 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 ): #isn't there a numpy function that does this and even gets called in Solver?  
 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  
 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 # Shouldn't it cost something to go downhill, no matter how steep??????  
  
  
 return int(math.ceil(cost \* self.MAP\_SCALE))

Proj5GUI.py

#!/usr/bin/env python3  
  
import math  
import random  
import signal  
import sys  
import time  
  
  
from which\_pyqt import PYQT\_VER  
if PYQT\_VER == 'PYQT5':  
 from PyQt5.QtWidgets import \*  
 from PyQt5.QtGui import \*  
 from PyQt5.QtCore import \*  
else:  
 raise Exception('Unsupported Version of PyQt: {}'.format(PYQT\_VER))  
  
  
#*TODO: Error checking on txt boxes*#*TODO: Color strings*# Import in the code with the actual implementation  
from TSPSolver import \*  
#from TSPSolver\_complete import \*  
from TSPClasses import \*  
  
  
class PointLineView( QWidget ):  
 def \_\_init\_\_( self, status\_bar, data\_range ):  
 super(QWidget,self).\_\_init\_\_()  
 self.setMinimumSize(950,600)  
  
 self.pointList = {}  
 self.edgeList = {}  
 self.labelList = {}  
 self.status\_bar = status\_bar  
 self.data\_range = data\_range  
 self.start\_pt = None  
 self.end\_pt = None  
  
 def displayStatusText(self, text):  
 self.status\_bar.showMessage(text)  
  
 def clearPoints(self):  
 self.pointList = {}  
  
 def clearEdges(self,removeColors = None):  
 self.edgeList = {}  
 if removeColors: # allows removal of edge labels without removing node labels, for example  
 for color in removeColors:  
 if color in self.labelList:  
 del self.labelList[color]   
 else:  
 self.labelList = {}  
 self.repaint()  
  
 def addPoints( self, point\_list, color ):  
 if color in self.pointList:  
 self.pointList[color].extend( point\_list )  
 else:  
 self.pointList[color] = point\_list  
  
# def setStartLoc( self, point ):  
# self.start\_pt = point  
# self.repaint()  
#  
# def setEndLoc( self, point ):  
# self.end\_pt = point  
# self.repaint()  
  
  
 def addEdge( self, startPt, endPt, label, edgeColor, labelColor=None, xoffset=0.0 ):  
 if not labelColor:  
 labelColor = edgeColor  
  
 assert( type(startPt) == QPointF )  
 assert( type(endPt) == QPointF )  
 assert( type(label) == str )  
  
 edge = QLineF(startPt, endPt)  
 if edgeColor in self.edgeList.keys():  
 self.edgeList[edgeColor].append( edge )  
 else:  
 self.edgeList[edgeColor] = [edge]  
  
 midp = QPointF( (edge.x1()\*0.2 + edge.x2()\*0.8),   
 (edge.y1()\*0.2 + edge.y2()\*0.8) )  
 self.addLabel( midp, label, labelColor, xoffset=xoffset )  
  
 def addLabel( self, point, label, labelColor,xoffset=0.0 ):  
 if labelColor in self.labelList.keys():  
 self.labelList[labelColor].append( (point,label,xoffset) )  
 else:  
 self.labelList[labelColor] = [(point,label,xoffset)]  
  
  
  
  
 def paintEvent(self, event):  
 painter = QPainter(self)  
 painter.setRenderHint(QPainter.Antialiasing,True)  
  
 xr = self.data\_range['x']  
 yr = self.data\_range['y']  
 w = self.width()  
 h = self.height()  
 w2h\_desired\_ratio = (xr[1]-xr[0])/(yr[1]-yr[0])  
 if w / h < w2h\_desired\_ratio:  
 scale = w / (xr[1]-xr[0])  
 else:  
 scale = h / (yr[1]-yr[0])  
  
 tform = QTransform()  
 tform.translate(self.width()/2.0,self.height()/2.0)  
 tform.scale(1.0,-1.0)  
 painter.setTransform(tform)  
  
 for color in self.edgeList:  
 c = QColor(color[0],color[1],color[2])  
 painter.setPen( c )  
 for edge in self.edgeList[color]:  
 ln = QLineF( scale\*edge.x1(), scale\*edge.y1(), scale\*edge.x2(), scale\*edge.y2() )  
 painter.drawLine( ln )  
  
 for color in self.edgeList:  
 c = QColor(color[0],color[1],color[2])  
 painter.setPen( c )  
 for edge in self.edgeList[color]:  
 #arrow\_scale = .015  
 arrow\_scale = 5.0  
 unit\_edge = ( edge.x2() - edge.x1(), edge.y2() - edge.y1() )  
 unit\_edge\_mag = math.sqrt( ( edge.x2() - edge.x1())\*\*2 + ( edge.y2() - edge.y1() )\*\*2 )  
 unit\_edge = (unit\_edge[0] / unit\_edge\_mag, unit\_edge[1] / unit\_edge\_mag )  
 unit\_edge\_perp = (-unit\_edge[1], unit\_edge[0])  
  
 temp\_tform = QTransform()  
 temp\_tform.translate(self.width()/2.0,self.height()/2.0)  
 temp\_tform.scale(1.0,-1.0)  
 temp\_tform.translate(scale\*edge.x2(),scale\*edge.y2())  
 temp\_tform.scale(1.0,-1.0)  
 painter.setTransform(temp\_tform)  
 #painter.drawText( RECT, label[1], align )  
  
 tri\_pts = []  
 tri\_pts.append( QPointF(0,0) )  
 tri\_pts.append( QPointF(-arrow\_scale\*(2\*unit\_edge[0] + unit\_edge\_perp[0]),  
 arrow\_scale\*(2\*unit\_edge[1] + unit\_edge\_perp[1])) )  
 tri\_pts.append( QPointF(-arrow\_scale\*(2\*unit\_edge[0] - unit\_edge\_perp[0]),  
 arrow\_scale\*(2\*unit\_edge[1] - unit\_edge\_perp[1])) )  
 tri = QPolygonF( tri\_pts )  
 b = painter.brush()  
 painter.setBrush( c )  
 painter.drawPolygon( tri )  
 painter.setBrush( b )  
  
 painter.setTransform(tform)  
 font = QFont("Monospace")  
 font.setStyleHint(QFont.TypeWriter)  
  
 R = 1.0E3  
 CITY\_SIZE = 2.0 # DIAMETER  
 RECT = QRectF(-R,-R,2.0\*R,2.0\*R)  
 align = QTextOption( Qt.Alignment(Qt.AlignHCenter | Qt.AlignVCenter) )  
 for color in self.labelList:  
 c = QColor(color[0],color[1],color[2])  
 painter.setPen( c )  
 for label in self.labelList[color]:  
 temp\_tform = QTransform()  
 temp\_tform.translate(self.width()/2.0,self.height()/2.0)  
 temp\_tform.scale(1.0,-1.0)  
 pt = label[0]  
 xoff = label[2]  
 temp\_tform.translate(scale\*pt.x()+xoff,scale\*pt.y())  
 temp\_tform.scale(1.0,-1.0)  
 painter.setTransform(temp\_tform)  
 painter.drawText( RECT, label[1], align )  
  
 painter.setTransform(tform)  
 for color in self.pointList:  
 c = QColor(color[0],color[1],color[2])  
 painter.setPen( c )  
 b = painter.brush()  
 painter.setBrush(c)  
 for point in self.pointList[color]:  
 pt = QPointF(scale\*point.x(), scale\*point.y())  
 painter.drawEllipse( pt, CITY\_SIZE, CITY\_SIZE)  
 painter.setBrush(b)  
  
  
  
class Proj5GUI( QMainWindow ):  
  
 def \_\_init\_\_( self ):  
 super(Proj5GUI,self).\_\_init\_\_()  
  
 self.RED\_STYLE = "background-color: rgb(255, 220, 220)"  
 self.PLAIN\_STYLE = "background-color: rgb(255, 255, 255)"  
 self.\_MAX\_SEED = 1000   
  
 self.\_scenario = None  
 self.initUI()  
 self.solver = TSPSolver( self.view )  
 self.genParams = {'size':None,'seed':None,'diff':None}  
  
  
   
 def newPoints(self):   
 # *TODO - ERROR CHECKING!!!!* seed = int(self.curSeed.text())  
 random.seed( seed )  
  
 ptlist = []  
 RANGE = self.data\_range  
 xr = self.data\_range['x']  
 yr = self.data\_range['y']  
 npoints = int(self.size.text())  
 while len(ptlist) < npoints:  
 x = random.uniform(0.0,1.0)  
 y = random.uniform(0.0,1.0)  
 if True:  
 xval = xr[0] + (xr[1]-xr[0])\*x  
 yval = yr[0] + (yr[1]-yr[0])\*y  
 ptlist.append( QPointF(xval,yval) )  
 return ptlist  
  
 def generateNetwork(self):  
 points = self.newPoints() # uses current rand seed  
 diff = self.diffDropDown.currentText()  
 rand\_seed = int(self.curSeed.text())  
 self.\_scenario = Scenario( city\_locations=points, difficulty=diff, rand\_seed=rand\_seed )  
  
 self.genParams = {'size':self.size.text(),'seed':self.curSeed.text(),'diff':diff}  
 self.view.clearEdges()  
 self.view.clearPoints()  
  
 self.addCities()  
  
  
  
 def addCities( self ):  
 cities = self.\_scenario.getCities()  
 self.view.clearEdges()  
 for city in cities:  
 self.view.addLabel( QPointF(city.\_x, city.\_y), city.\_name, \  
 labelColor=(128,128,128), xoffset=10.0 )  
  
 def generateClicked(self):  
 self.generateNetwork()  
 self.view.addPoints( [QPointF(c.\_x,c.\_y) for c in self.\_scenario.getCities()], (0,0,0) )  
 self.solveButton.setEnabled(True)  
 self.graphReady = True  
 self.checkGenInputs()  
 self.numSolutions.setText( '--' )  
 self.tourCost.setText( '--' )  
 self.solvedIn.setText( '--' )  
 self.maxQSize.setText( '--' )  
 self.totalStates.setText( '--' )  
 self.prunedStates.setText( '--' )  
 self.statusBar.showMessage('')  
 self.view.repaint()  
  
  
 def displaySolution( self ) : # what about calling this somehow every time a new bssf is found?  
 self.view.clearEdges([(64,64,255)]) # get rid of edge labels but not point labels  
 if self.\_solution:  
 self.addCities()  
 edges = self.\_solution.enumerateEdges()  
 if edges:  
 edgeColor = (128,128,255)  
 labelColor = (64,64,255)  
 for edge in edges:  
 pt1,pt2,label = edge  
 self.view.addEdge( QPointF(pt1.\_x,pt1.\_y), \  
 QPointF(pt2.\_x,pt2.\_y), \  
 '{}'.format(label), edgeColor, labelColor )  
 else:  
 self.statusBar.showMessage('No Solution Found.')  
 self.view.repaint()  
  
  
 def randSeedClicked(self):  
 new\_seed = random.randint(0, self.\_MAX\_SEED-1)  
 self.curSeed.setText( '{}'.format(new\_seed) )  
 self.view.repaint()  
  
 def solveClicked(self): # need to reset display??? and say "processing..." at bottom???  
 self.solver.setupWithScenario(self.\_scenario)  
  
 max\_time = float( self.timeLimit.text() )  
 # *TODO - start on a separate thread* self.view.clearEdges([(64,64,255)]) # get rid of edge labels but not point labels  
 self.numSolutions.setText( '--' )  
 self.tourCost.setText( '--' )  
 self.solvedIn.setText( '--' )  
 self.maxQSize.setText( '--' )  
 self.totalStates.setText( '--' )  
 self.prunedStates.setText( '--' )  
 self.statusBar.showMessage('Processing...')  
 #self.view.repaint()  
 #app.processEvents()  
 solve\_func = 'self.solver.'+self.ALGORITHMS[self.algDropDown.currentIndex()][1]  
 results = eval(solve\_func)(time\_allowance=max\_time )  
 if results:  
 self.statusBar.showMessage('')  
 self.numSolutions.setText( '{}'.format(results['count']) )  
 self.tourCost.setText( '{}'.format(results['cost']) )  
 self.solvedIn.setText( '{:6.6f} seconds'.format(results['time']) )  
 self.\_solution = results['soln']  
 if 'max' in results.keys():  
 self.maxQSize.setText( '{}'.format(results['max']))  
 if 'total' in results.keys():  
 self.totalStates.setText( '{}'.format(results['total']))  
 if 'pruned' in results.keys():  
 self.prunedStates.setText( '{}'.format(results['pruned']))  
 #if self.\_solution:  
 self.displaySolution()  
 else:  
 print( 'GOT NULL SOLUTION BACK!!' ) #probably shouldn't ever use this...  
 self.view.repaint()  
# app.processEvents()  
  
 def checkGenInputs(self):  
 seed = self.curSeed.text()  
 size = self.size.text()  
 diff = self.diffDropDown.currentText()  
  
 if self.\_scenario:  
 if self.genParams['seed'] == seed and \  
 self.genParams['size'] == size and \  
 self.genParams['diff'] == diff:  
 self.generateButton.setEnabled(False)  
 self.solveButton.setEnabled(True)  
 elif (seed == '') or (size == ''):  
 self.generateButton.setEnabled(False)  
 self.solveButton.setEnabled(True)  
 else:  
 self.generateButton.setEnabled(True)  
 self.solveButton.setEnabled(False)  
  
  
 def checkInputValue(self, widget, validrange):  
 assert( type(widget) == QLineEdit )  
 retval = None  
 valid = False  
 try:  
 sval = widget.text()  
 if sval == '':  
 valid = True  
 else:  
 ival = int(sval)  
 if validrange:  
 if ival >= validrange[0] and ival <= validrange[1]:  
 retval = ival  
 valid = True  
 except:  
 pass  
  
 if not valid:  
 widget.setStyleSheet( self.RED\_STYLE )  
 else:  
 widget.setStyleSheet( '' )  
  
 return '' if retval==None else retval  
   
 ALGORITHMS = [ \  
 ('Default ','defaultRandomTour'), \  
 ('Greedy','greedy'), \  
 ('Branch and Bound','branchAndBound'), \  
 ('Fancy','fancy') \  
 ] # whitespace hack to get longest to display correctly  
  
 def initUI( self ):  
 self.setWindowTitle('Traveling Salesperson Problem')  
 self.setWindowIcon( QIcon('icon312.png') )  
  
 self.statusBar = QStatusBar()  
 self.setStatusBar( self.statusBar )  
  
 vbox = QVBoxLayout()  
 boxwidget = QWidget()  
 boxwidget.setLayout(vbox)  
 self.setCentralWidget( boxwidget )  
  
  
 SCALE = 1.0  
 self.data\_range = { 'x':[-1.5\*SCALE,1.5\*SCALE], \  
 'y':[-SCALE,SCALE] }  
 self.view = PointLineView( self.statusBar, \  
 self.data\_range )  
 self.randSeedButton = QPushButton('Randomize Seed')  
 self.generateButton = QPushButton('Generate Scenario')  
 self.solveButton = QPushButton('Solve TSP')  
  
 self.curSeed = QLineEdit('20')  
 self.curSeed.setFixedWidth(100)  
 self.size = QLineEdit('15')  
 self.size.setFixedWidth(50)  
 self.timeLimit = QLineEdit('60')  
 self.timeLimit.setFixedWidth(50)  
 self.numSolutions = QLineEdit('--')  
 self.numSolutions.setFixedWidth(100)  
 self.tourCost = QLineEdit('--')  
 self.tourCost.setFixedWidth(100)  
 self.solvedIn = QLineEdit('--')  
 self.solvedIn.setFixedWidth(200)  
 self.maxQSize = QLineEdit('--')  
 #self.maxQSize.setFixedWidth(100)  
 self.totalStates = QLineEdit('--')  
 #self.totalStates.setFixedWidth(200)  
 self.prunedStates = QLineEdit('--')  
 #self.prunedStates.setFixedWidth(200)  
  
 self.diffDropDown = QComboBox(self)  
 self.algDropDown = QComboBox(self)  
  
 h = QHBoxLayout()  
 h.addWidget( self.view )  
 vbox.addLayout(h)  
  
 h = QHBoxLayout()  
 h.addStretch(1)  
 h.addWidget( QLabel( 'max queue size:' ) )  
 h.addWidget( self.maxQSize )  
 self.maxQSize.setEnabled(False)  
 vbox.addLayout(h)  
  
 h = QHBoxLayout()  
 h.addStretch(1)  
 h.addWidget( QLabel( 'total states:' ) )  
 h.addWidget( self.totalStates )  
 self.totalStates.setEnabled(False)  
 vbox.addLayout(h)  
  
 h = QHBoxLayout()  
 h.addStretch(1)  
 h.addWidget( QLabel( 'pruned states:' ) )  
 h.addWidget( self.prunedStates )  
 self.prunedStates.setEnabled(False)  
 vbox.addLayout(h)  
  
  
 h = QHBoxLayout()  
 h.addWidget( QLabel('Problem Size: ') )  
 h.addWidget( self.size )  
 h.addWidget( QLabel('Difficulty: ') )  
 h.addWidget( self.diffDropDown )  
 h.addWidget( QLabel('Current Seed: ') )  
 h.addWidget( self.curSeed )  
 h.addWidget( self.randSeedButton )  
 h.addWidget( self.generateButton )  
 h.addStretch(1)  
 vbox.addLayout(h)  
   
 h = QHBoxLayout()  
 h.addWidget( QLabel('Algorithm: ') )  
 h.addWidget( self.algDropDown )  
 h.addWidget( QLabel( 'Time Limit' ) )  
 h.addWidget( self.timeLimit )  
 h.addWidget( QLabel( 'seconds' ) )  
 h.addWidget( self.solveButton )  
 h.addStretch(1)  
 vbox.addLayout(h)  
  
 h = QHBoxLayout()  
 h.addWidget( QLabel( '# Solutions:' ) )  
 h.addWidget( self.numSolutions )  
 h.addWidget( QLabel( 'Cost of tour:' ) )  
 h.addWidget( self.tourCost )  
 h.addWidget( QLabel( 'Solved in:' ) )  
 h.addWidget( self.solvedIn )  
 self.numSolutions.setEnabled(False)  
 self.tourCost.setEnabled(False)  
 self.solvedIn.setEnabled(False)  
 h.addStretch(1)  
 vbox.addLayout(h)  
  
  
 self.lastPath = (None,None)  
 self.solveButton.setEnabled(False)  
  
 self.curSeed.textChanged.connect(self.checkGenInputs)  
 self.size.textChanged.connect(self.checkGenInputs)  
  
 self.randSeedButton.clicked.connect(self.randSeedClicked)  
 self.generateButton.clicked.connect(self.generateClicked)  
 self.solveButton.clicked.connect(self.solveClicked)  
  
 self.diffDropDown.addItem('Easy ') # Weird hack to make box wide enough to show all of last item  
 self.diffDropDown.addItem('Normal')  
 self.diffDropDown.addItem('Hard')  
 self.diffDropDown.addItem('Hard (Deterministic)')  
 self.diffDropDown.activated.connect(self.diffChanged)  
 self.diffDropDown.setCurrentIndex(3)  
 self.diffChanged(3) # to handle start state  
  
 for alg in self.ALGORITHMS:  
 self.algDropDown.addItem( alg[0] )  
 self.algDropDown.activated.connect(self.algChanged)  
 self.algDropDown.setCurrentIndex(2)  
 self.algChanged(2) # to handle start state  
  
 self.graphReady = False  
  
 self.show()  
  
  
 def diffChanged(self, text):  
 self.checkGenInputs()  
  
 def algChanged(self, text):  
 pass  
  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 # This line allows CNTL-C in the terminal to kill the program  
 signal.signal(signal.SIGINT, signal.SIG\_DFL)  
   
 app = QApplication(sys.argv)  
 w = Proj5GUI()  
 sys.exit(app.exec())

which\_pyqt.py

PYQT\_VER = 'PYQT5'