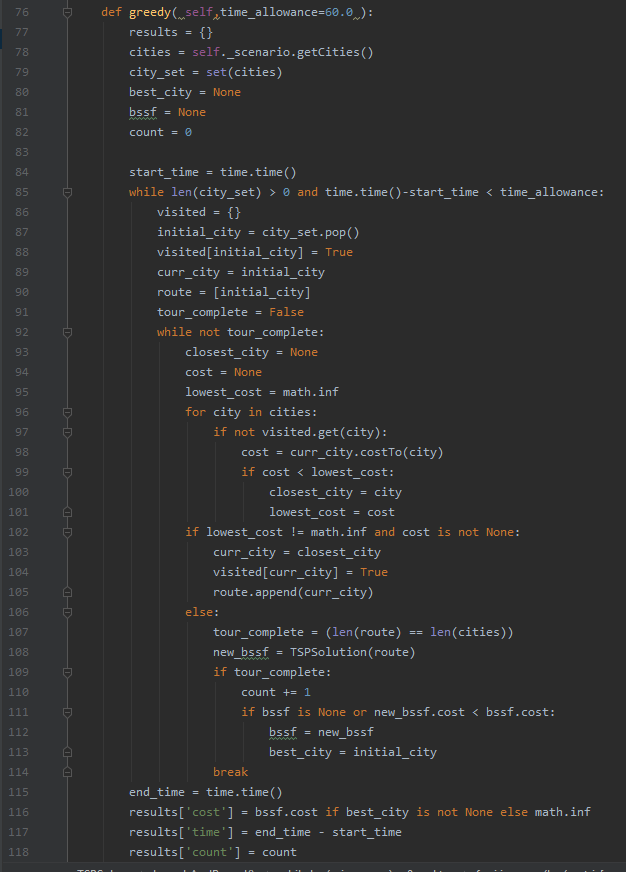
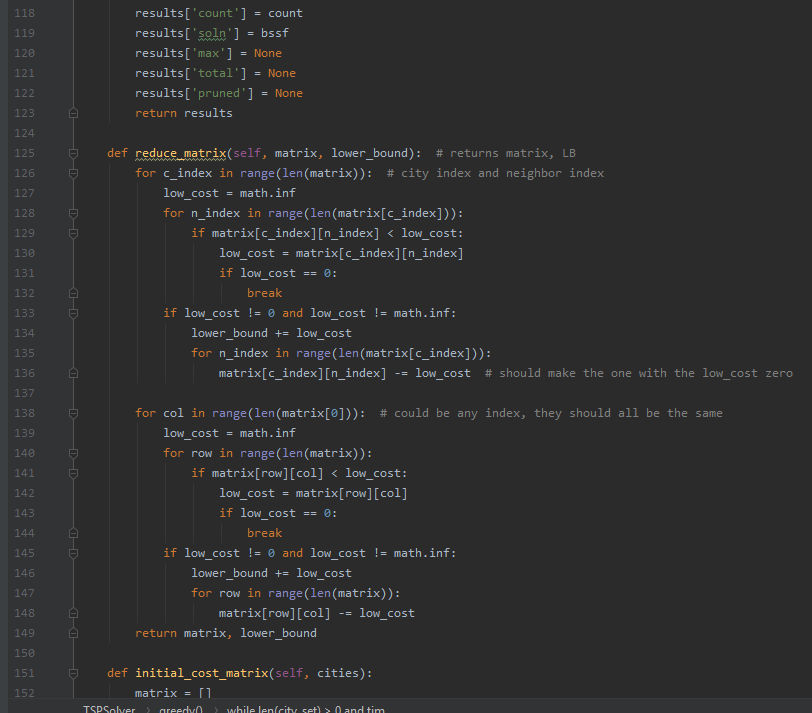
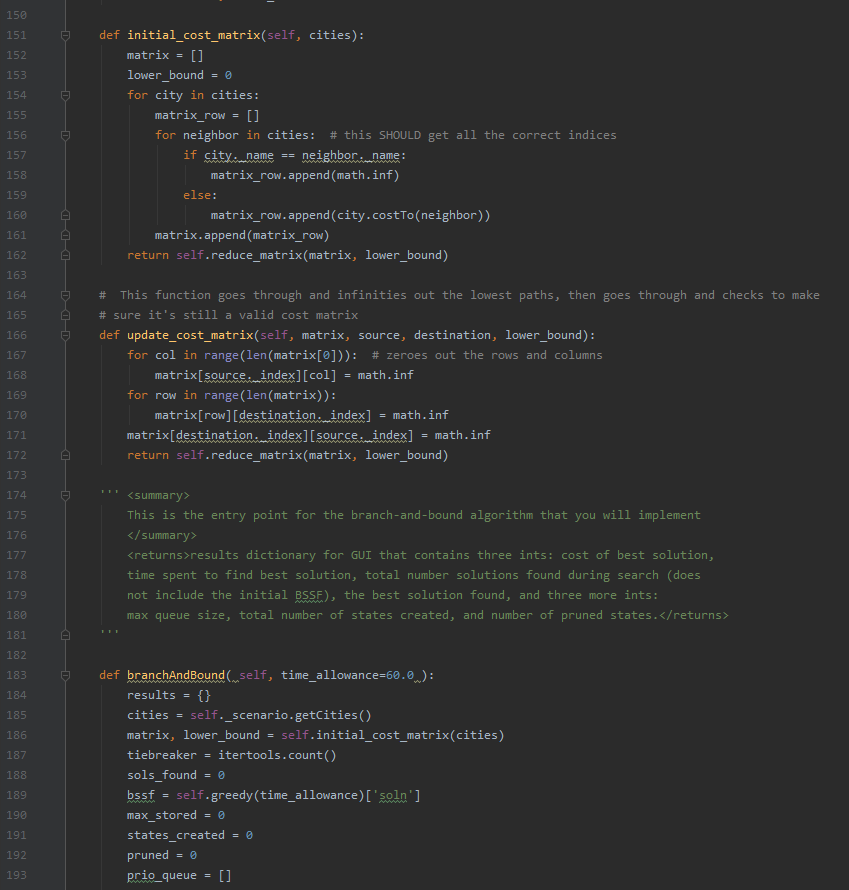
Taylor Whitlock

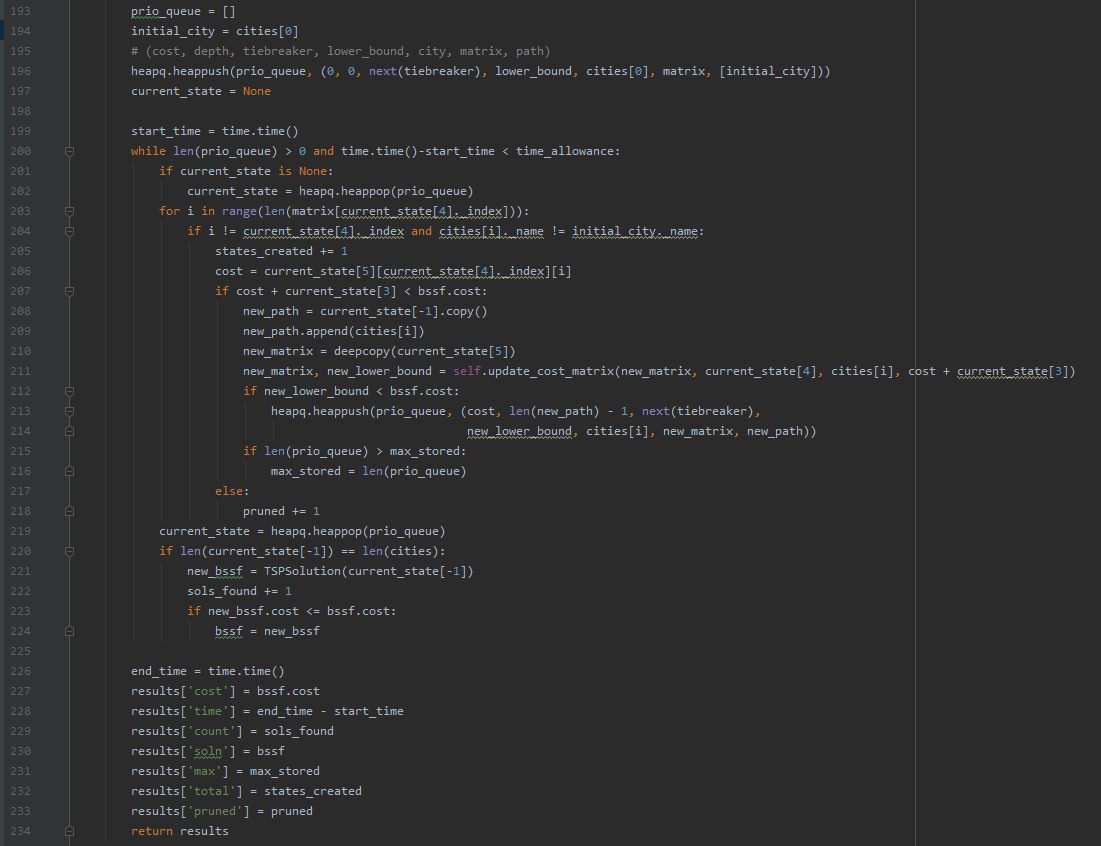
Project 5: TSP

Part 1:









Part 2:

I used my greedy algorithm for the initial BSSF, so the time/space analysis of that should be included here, I think.

In terms of time complexity, the cost to make the initial set of cities would just be O(n), as it would have to go through each element and add it to a set. The outermost while loop is also O(n) since it just goes through each city in the set.

Afterwards, the inner while loop is roughly O(n), but could be less since it breaks once a complete tour is found. Inside that while loop is a for loop iterating through each city and checking if it’s visited, another O(n) loop, making the grand total for time complexity O(n^3).

The space complexity is O(n) for the initial set, plus O(n) for the route, plus O(n) for the dictionary of whether the city has been visited or not, making the overall asymptotic space complexity O(n).

The initial cost for the BSSF of the Branch and Bound algorithm is the cost for greedy, since that is how I got my initial BSSF, so we start with O(n^3) time and O(n) space complexity.

The cost to reduce the RCM at any time is O(n^2), since it has to go through each row and column, determine the lowest cost there, and subtract that low cost from each element. Reducing the rows/columns separately are both technically O(2n^2) and combined are O(4n^2) but asymptotically just O(n^2). Nothing new is created in the reduction, so the space complexity I believe is just O(n^2) from the matrix that is passed in.

The initial cost matrix is simply the cost of reduction plus the initial creation, therefore O(n^2) time complexity as well as O(n^2) space complexity.

The cost of updating the matrix is the cost of reduction plus O(2n) for looping through a single row and a single column to “infinity them out,” so that future child states won’t try to return to a node that has been visited already. That results in the overall asymptotic time complexity of updating being O(n^2). Since every update copies the matrix. It adds to the space complexity by a factor of O(n^2).

For time complexity of the overall branch and bound algorithm, it is the cost of the initial matrix creation plus the cost of the initial bssf(O(n^3) + O(n^2)), plus the cost of the following:

* Insertion/deletion from the priority queue (using heapq) is O(logn)
* The outermost while loop that is looping until the priority queue is empty will be something around O(2^n) because of pruning shenanigans.
* Looping through each city to create states is O(n), making the BnB algorithm alone a total of O(n^2) so far
* Copying the path adds a factor of O(n)
* Copying the matrix adds a factor of O(n^2) time and space complexity (space complexity already accounted for in the updating portion)
* Updating adds in O(n^2), explained above.

All of these together make a total time complexity of roughly O((2^n)\*(n^3)). I think the absolute worst case scenario (if NOTHING gets pruned) is O((n^n)\*(n^3)), since the priority queue while loop would have to run an exponential amount of times.

The space complexity of the algorithm alone is the size of the priority queue \* the size of the elements stored in it. Each element is a tuple with 4 ints, a city, the O(n^2) matrix, and the O(n) path, so each tuple is roughly O(n^2) in size, making the space complexity roughly O(n^3) if my assumption about the priority queue loop is correct.

Part 3:

For each state, I used a tuple structured like the following: (cost, depth, tiebreaker, lower\_bound, city, matrix, path). Cost, depth, tiebreaker, and lower\_bound were all ints, the city was a TSPClasses.City, the matrix was a 2D list of ints(the costs, really the RCM), and the path was a 1D list of cities.

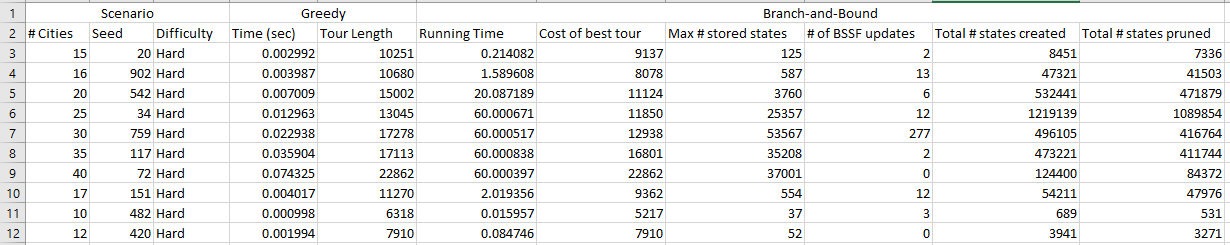
Part 4:

I used heapq for my priority queue, sorted by the cost of getting to that particular state, then by depth, then by a tiebreaker. Upon reflection it likely would’ve been better to sort by a combination of cost and depth, so that states closer to the solution had priority. With that implementation, it would probably prune a lot more states and thus create far fewer states. It is simply a min-heap queue storage system.

Part 5:

My approach for the initial BSSF was simply to use the greedy algorithm, since it tends to be quite fast but also gets a pretty decent solution to begin with.

Part 6:



Part 7:

I find it very interesting that with my data, the largest space required was for a graph of size 25. I assume that was because the greedy algorithm found an unusually large initial solution, so the branch and bound had to check and create more states to find increasingly better solutions. When it got to size 40, my guess is that it didn’t update the bssf because it ran out of time. That probably could have been improved with a slightly different implementation of how the heap was sorted.

The time complexity definitely seemed to fit the exponential growth, though I think because of the pruning it was still able to find good solutions.

The number of states pruned seemed to be about the same proportionally for each test, no matter the problem size.