TSP Project 5 Report

2) Time Complexity: Each iteration the algorithm must do the following procedures:

- Insert/Delete/Pop from Priority Queue: O(logn), O(logn), O(1)
- Logical Checks (exceeded time/is new BSSF): O(1)
- Generate list of valid paths to check: O(n)
- Update Adjacency Matrix: O(n^2) Since possibly near all of the values can be updated in the n*n matrix on a given pass.
- Generate other info for new subproblem node: O(1)
  Therefore the overall time complexity of a given iteration is **O(n^2)**

There are up to (n-1)! Potential worst-case iterations when no branches are pruned. Therefore, the overall worst-case time complexity is **O(n!)** yet the Branch-and-Bound technique reduces this to something far more manageable.

Space Complexity: There are multiple elements that need to be stored for each node in the problem tree. I have included those calculations in Question 3. Its overall complexity is of O(n^2) for one iteration. At any given time it can hold up to, in the absolute worst cast (n-1)! iterations. Therefore, the overall worst-case space complexity is O(n!). Again, the branch and bound technique reduces this real average space complexity significantly.

3) I represented each state as a tuple containing the following information:

- A PriorityCount object (Currently contains two values, the current lower bound and the order in which it was added. Since this is the first element Python's heapq library interprets it as a the 'key' of the tuple and uses it to balance the heap. The __lt__ comparison method has been overloaded to return the object with the lowest lower bound multiplied by the number of cities visited and, in the case of a tie, return the one added most recently as it is more likely to have explored more of the tree. I've structured it in this way to make different preferences easy to plug in later) **O(1)**
- A City object which represents the current root of the subtree to analyze. **O(1)**
- A Boolean list of the unvisited cities. This represents all of the direct ancestor nodes of the current root of the given subtree and is the same length as the total number of cities. (While this could be extrapolated out of the last element of the tuple containing a list of visited cities including it saves computation at the expense of a little space) **O(n)**
- A city adjacency matrix with all of the direct ancestor's changes. This is inherently of size n*n where n is the number of cities. **O(n^2)**
- A growing list of the path taken, when it's length is equal to the length of total cities it is a potential candidate for the new BSSF. **O(n)**

4) Python's heapq data structure is a priority queue built as a heap. This means that insertions, deletions run in O(logn) time while time to find the minimum value runs in O(1) time. In a priority queue some sort of property is assigned (either implicitly by the data structure or explicitly by declaring it on insertion) and is placed in the queue according to the value of that property in comparison to all others. As a queue, the only interactions a client has with it is pushing and popping. Popping will return the node of most priority as defined and pushing will insert a new value according to that given property.

5) To get my initial BSSF I have implemented the greedy algorithm approach. This gets the cost of all immediate unvisited cities for which a path exists and takes the one of lowest cost successively until all nodes have been explored. In a direct accuracy vs. time comparison it is an effect way to get the initial BSSF as it will disregard many inefficient routes near instantaneously.

7) Regarding the table I found it interesting how I was able to come up with a solution for the 16-city route in the nick of time but not for the 15 city route. While they could be more efficient I feel like the times are generally pretty good. As it is now I am weighing breadth more significant than depth which for me generally ended up with faster results as apposed to a more depth leaning approach. (In other words, the "weight" is calculated by the path cost multiplied by the number of cities in the path. This makes the impact of immediate good paths more evident but updates of the BSSF more infrequent). With occasional rare exceptions, like the first two, the run time grew as a steady function of the number of input cities. While it certainly isn't a factorial growth it is fairly rapid. It is important to note that the other features, such as the number of observed BSSF updats, the max number of stored states and the total # of created states all grow similarly, not factorial but very quickly.