# Lab 5 Report

1. [20 points] Well-documented code.
2. [10 points] An explanation of the time complexity of the algorithm by showing and summing up the complexity of each subsection of your code. Keep in mind the following things:
    - Priority Queue
        1. O(logn) push and pop
    - Search space
        1. O(N-1) + O(N-1) N times so O(N^2)
    - Reduced cost matrix and updating it
        1. O(N)
    - BSSF Initialization
        1. O(N^2)
    - Expanding one search state (subproblem) into others
        1. O(N) - constrained by reducing a matrix
    - The full branch and bound algorithm
        1. O(N^2)
3. [10 points] Describe the data structures you use to represent the search states (subproblems)
    i. The search states are represented as numpy matrices for the edges, an integer for the bound, and a list of integers for the route
    ii. I combined these three parameters into a class called PartialSolution
4. [5 points] Describe the priority queue data structure you use and how it works (you can use the implementation you built for lab 3 or you can use a python library such as bheap)
    i. I used the python library heapq. The priority queue held the bound of the partial solution. A dictionary with the bound as the key mapped the bound to a PartialSolution object so I could load popped parent states from the queue.
5. [5 points] Describe your approach for the initial BSSF (best solution so far)
    i. The initial BSSF is a matrix containing all zeros, the bound is infinity, and the route is empty. The first real BSSF comes from reaching a leaf in the search tree. We prune the priority queue of any potential solutions that have bounds higher than the BSSF bound.
6. [30 points] Include a table containing the following columns:

| # Cities | Seed | Running time (sec) | Cost of best tour * = optimal | Max number of stored states at a given time | # of BSSF updates | Total # of states created | Total # of states pruned |
|---|---|---|---|---|---|---|---|
| 15 | 20 | .761097 | 10534 | 3751 | 1 | 4352 | 3751 |
| 16 | 902 | 1.511625 | 8001 | 6945 | 2 | 8063 | 6946 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 10 | 5 | .168231 | 9207 | 854 | 2 | 854 | 855 |
| 18 | 40 | 7.447745 | 10526 | 33455 | 2 | 38844 | 33456 |
| 20 | 30 | 1.480910 | 11591 | 5640 | 2 | 6360 | 5641 |
| 18 | 7 | 3.136885 | 10413 | 11702 | 2 | 13950 | 11703 |
| 20 | 200 | 60 | 25011 | 252639 | 1 | 252639 | 0 |
| 30 | 31 | 60 | 40154 | 193919 | 1 | 193919 | 0 |
| 100 | 23 | 60 | 120778 | 50317 | 1 | 50317 | 0 |
| 50 | 99 | 60 | 64367 | 125650 | 1 | 125650 | 0 |

7. [10 points] Discuss the results in the table and why you think the numbers are what they are, including how time complexity and pruned states vary with problem size.

The number of BSSF updates is low because my algorithm does more of a breadth first approach versus a depth first approach. However, it appears to arrive at the the optimal solution in a fewer number of BSSF updates. The time it takes to complete is pretty small but doesn't work well past 20 items. The time does increase with respect to increase number of cities and complexity.