COSC 3020
Algorithms

Name: Jacob Tuttle

Lab 05

October 17, 2019

1. Graph Representations: Solutions to the permutation sort problem are provided within the lab file `lab_05_tuttle.html`.

2. Runtime Analysis: Analyzing the runtime of this algorithm is more straightforward than the run times for previous labs. First, let us break down our actions in plain english. Our algorithm will iterate over each possible connection of two vertices and check to see if it is a valid edge. If it is, then the edge is added to the `adjList` structure. If it is not, the algorithm just continues to check vertex pairs.

Examining the code:

```
for (var row = 0; row < adjMatrix.length; row++) {
    var tmp = [];
    for (var col = 0; col < adjMatrix[row].length; col++) {
        if (adjMatrix[row][col] == 1) {
            tmp.push(col);
        }
    }
    adjList.push(tmp);
}
```

we can see that each of the for loops will execute $O(V)$ times, making the overall nested for loop complexity $O(V^2)$. The only other operations performed by this algorithm are pushing the found edges onto `adjList`. This action will take $O(E)$ since the action will be performed once for each edge. This gives a final time complexity for our algorithm of $O(E + V^2)$.

Since the number of edges and vertices can change depending on the graph, and they vary such that one could be more influential than another on the overall time complexity, both of them are included in the big O notation. In regards to a function that converts in the opposite direction, this

could execute in linear time with relation to the number of edges, $O(E)$, since javascript can create a two-dimensional array in constant time. This algorithm would simply iterate through the edges and add them to the ajacency matrix one after the other.