Youssef Naguib <ymn4543@rit.edu>
CS261
Write-up for hw-5 questions 1-4

1. **Correctness argument:**
   Given a M x N maze, my algorithm finds the minimum number of steps required to reach a destination node from a start node. The algorithm starts from the start coordinate given by input, and uses a Breadth first search to access all possible neighbors, storing them in a queue data structure. This process is repeated, and each "depth level" of the search is considered to be a single move. The algorithm only checks moves possible in the north, south, west, and east directions, and stops once it reaches the destination coordinate. Since it is a breadth first search, we know that the number of layers in the search up to the destination node is guaranteed to be the minimal number of steps required to complete the maze.

   Running Time Estimate:
   1. Check directions - O(1)
   2. Run BFS – O(N+M)
   3. Return answer – O(1)

      Total = O(N+M)

2. **Correctness argument:**
   This problem took much thought and effort, but I managed to break it up into steps so that I could logically make a decision on whether a graph could become strongly connected by adding one single edge. It begins by reading in the input of course, and constructing an adjacency list. Using this adjacency list, we run the strongly connected components algorithm to compute the number of strongly connected components and their corresponding vertices. We can now simplify this graph into a directed acyclic graph (DAG) by minimizing each strongly connected component into one vertex. My construct_DAG() algorithm does this and builds an adjacency matrix out of the minimized graph. At first I was attempting to construct an adjacency list, but this proved to be very inefficient for large inputs, as checking if an element was already in the list was linear time, compared to the constant time operations of a matrix.

   After we have the completed matrix for the DAG, we can then compute the minimal number of edges we need to convert the graph into a strongly connected graph. The logic behind this is as follows: we find the number of vertices in the DAG with no incoming edges (an in-degree of 0), and then we find the number of vertices with no outgoing edges (an out-degree of 0). The answer will always be the maximum of these two values. I argue this because if there is ever a vertex with no outgoing edges or a vertex with no incoming edges, the graph cannot possibly be strongly connected, since one vertex will not be mutually reachable. So we can conclude that if the maximum of these two values is equal to one, then and only then can we convert the graph to be strongly connected with the addition of a single edge.

   Running Time Estimate:
   1. Find strongly connected components – O(N + M)
   2. Construct DAG – O(N + M)
   3. Find number of edges needed – O(N + M)

      Total = O(3N + 3M) = O(N+M)

3.

Correctness argument:

This problem was quite trivial. We have a set of edges F that contains edges that must be included in the spanning tree of a graph G. To find the minimal spanning tree of G that included all edges in F, I used a modified version of Kruskal's algorithm. The algorithm uses a union-find data structure to ensure no cycles are created. The only difference between my version is that before sorting the edges and running the algorithm from the cheapest edge onwards, I add all the edges in F to the final spanning tree, and the union-find data structure, and then remove them from the total set of edges. From then on, the algorithm runs normally, sorting all edges and taking the minimal cost edges without creating a cycle, until a completed spanning tree is formed. Since the edges in F are all added first, this ensures that the final spanning tree is an F-containing spanning tree, and that it is the cheapest possible one, since the edges are sorted.

Running Time Estimate:

# of vertices = N, # of edges = M

1. Initialize Union-Find – O(N)
2. Union find operations on F edges – O(N*logN)
3. For all M edges: Do union-find operations – O(M*LogN)

Total = O(M*logN)

4.

Correctness argument:

This problem is identical to the shortest path problem that the classic Dijkstra algorithm solves, except the graph's vertices are weighted rather than it's edges. To account for this difference, a few minor adjustments had to be made to the classic Dijkstra algorithm. I store the weights for all vertices by their index in an array. For example the weight of vertex 1 would be at index 0 in the array. The update method inside of the Dijkstra algorithm now updates weights based on vertex weights not edge weights. The starting vertex has its weight set to its corresponding weight rather than 0. Other than that, the Dijkstra algorithm runs normally and keeps track of visited vertices. The minimal distance to each node in the graph is calculated and outputted to the user.

Running Time Estimate:

1. Iterate over each vertex – O(N)
2. Update the weights – O(N)
3. Remove the vertex from unvisited array – O(1)

Total = O(N^2)