Task 1 @

For adjacency matrix, I created a function that takes the first two elements of a line to designate the coordinates of where the third element (weight) will be placed. while the matrix dimension is (vertices +1) x (vertices +1).

Task 2 & 1 (b)

Here I take a dictionary that has a key for every vertex. Then in every input line I use the first element to determine the key, and add the other two elements as tuple. Finally print the dictionary.

Task 2:

The bfs function takes the adj list, visited and queue array. Adding the first element to the queue, we enter a while loop. Here if the node is unvisited, it prints, then it goes to its immediate neighbors and add them to queue. These then are

popped and prited if unvisited. The whole function is under a wrapper which ensures bfs recurs till all nodes are visited.

Task 3:

dfs is a recursive function that recurs through a path till it is fully explored, then comes back and does the same for the next element in the loop. It is also under a wrapper that makes dfs run till all nodes are visited.

Task 4:

To find a cycle, we modify the dfs function. This takes an additional path array, which is operated simultaneously with the visited array. The 'cycle or cycler( )' line notes the final answer. The path array indexes will be turned false again, once a path is fully visited. If while visiting a path, the corresponding

path array element is true, the we have a cycle, otherwise not.

Task 5:

To find the shortest path to a node from source, we run the most basic bfs but with and extra parent array. This array stores the immediate parents of each node. Then in the pathcheck() function, we backtrak the target element to the source, while adding them in a stack. Then printing the stack gives us the answer.

Task 6:

we first declare a list that contains the four possible places to go from a coordinate, Path. we make a new adjacency matrix, add a valid() function. Now run bfs. After

appending the first element in the queue, marked it visited. We take a count variable to count accumulated diamonds. Now inside the while loop, we unpack the coordinate tuple and check if it's a diamond and count accordingly. Then the inner loop runs on the path array, generating the four spaces around the given space. After each space is created, it runs through the valid function. This function checks whether the coordinate is a wall ('#') or the index goes out of range. Valid spaces are pushed in queue. (valid and not visited). Then there is a global counter list that store the number of diamonds in every bfs run. simply return the max value in that list to get the answer.