NABONITA SAHA (22301645).

Task 1(a).

First we create an array of (vertice+1) × (vertice+1). size. Then through ~~nested~~, looping we read each line and place the value of 'w' (weight) in the u th row and v th column of the array replacing the zero. Then we perform a nested looping in order to generate the output in the file.

Task 1(b)

We use the 'arr' array from Task 1(a) to perform this task. We performed a nested loop where it checked if the entry of the array was not 0. If it was not 0, the node and its weight was stored in the ~~list~~ 'new' ~~array~~. Then through further iteration on ~~the~~ 'new' - string was printed for each of the element and their linked nodes along with their weight in tuple form.

Task 2

BFS (Breadth First ~~traversal~~ Search) traversal uses Queue and adjacency list for its convenience. First we created an adjacency list that we stored in a form of dictionary. We took the starting node as ~~s~~ '1' and took visited as a set() since it takes all unique elements only. We appended the first node in 'Queue' list. Each time, we took the first element of the Queue to be the current node and added it in 'visited'. Then we performed a looping where if the current nodes adjacent element were not in "visited", we appended them in the queue and considered them to be the ~~next~~ visited nodes. We made sure the current nodes were being generated in the output.

Task 03

DFS traversal works on with the help of stack implementation. First we created an adjacency list and appended the first node inside 'stack1'. Inside a conditional loop, each time we considered the last element of the stack to be the 'current node' and if it hadn't been visited we generated it as the output for that iteration and added it within the 'visited' list. Then for that current node, we looked for its adjacent elements and if they were not already visited, we appended them within the 'Stack1'.

Task 04 :

Based on the directed graph, we created an adjacency list at first. We at first keep cycle as a flag. Then for each city in the adjacency list, if the city isn't visited we call the 'dfs traversal' function that returns True if the city is already in 'stack1' and returns False if city is present in 'visited', else they add the city in respective sets. This further leads to checking the adjacent cities as well and through further conditioning it determines whether cycle will exist or not. If cycle = True, it gives 'YES' and vise versa.

Task 05

Shortest path is easier to be found through the idea of BFS traversal as it covers a lot of area within a short amount of time. First we create an adjacency list. We ~~sto~~ take the starting node to be '1' and the end node is also defined. Through a function we pass the 'start' and 'end' nodes as the parameters and until the queue is empty, we check whether the last node of our "curr/path" ~~or~~ is the final destination or not. We further append the adjacent within the queue for that node and only return the path if its equal to the 'final'.

Task 06

Here, I created a list 'list1' that formed list elements of each characters of the input within lists. Then I created another list that marked '#' as 'T' since those cannot be traversed. Then in order to find the "maxcount" we ~~travers~~ perform a nested loop that calls a function "Travel" if the element of list1 is equivalent to '.' and that of 'new' in that index is 'O'. Here the traversing function works based on "dfs" where it checks on the elements left, right, up and down. It then recursively calls the function for the next element.