Code :

```
"""
You can create any other helper funtions.
Do not modify the given functions
"""

class Graph :

    def __init__(self, cost, start, goals, heuristic = []):

        self.cost = cost # 2D list
        self.heuristic = heuristic # 1D list
        self.start = start # integer
        self.goals = goals # 1D list
        self.pathCost = 0

    def aStar(self) :

        path = []
        explored = []

        path = [self.start]
        frontier = [[0 + self.heuristic[self.start], path]]
        while len(frontier) > 0:
            curr_cost, curr_path = frontier.pop(0)
            n = curr_path[-1]
            curr_cost -= self.heuristic[n]
            if n in self.goals:
                return curr_path
            explored.append(n)
            children = [i for i in range(len(self.cost[0]))
                        if self.cost[n][i] not in [0, -1]]
            for i in children:
                new_curr_path = curr_path + [i]
                new_path_cost = curr_cost + self.cost[n][i] + self.heuristic[i]
                if i not in explored and new_curr_path not in [i[1] for i in
frontier]:
                    frontier.append((new_path_cost, new_curr_path))
                    frontier = sorted(frontier, key=lambda x: (x[0], x[1]))
                elif new_curr_path in [i[1] for i in frontier]:
                    index = search_q(frontier, new_curr_path)
                    frontier[index][0] = min(frontier[index][0],
new_path_cost)
```

```
                frontier = sorted(frontier, key=lambda x: (x[0], x[1]))

        return list

    def dfs(self) :

        path = []
        stack = [self.start]
        visited = set()

        while len(stack):
            current_node = stack.pop()
            if current_node not in visited:
                visited.add(current_node)
                path.append(current_node)

            if current_node in self.goals:
                return path
            no_neighbour = 1
            for neighbour in range(len(self.cost) - 1, 0, -1):
                if neighbour not in visited and self.cost[current_node]
[neighbour] > 0:
                    stack.append(neighbour)
                    no_neighbour = 0
            if no_neighbour and len(path):
                stack.append(path[-1])
                children = [i for i in range(len(cost)) if i not in visited and
cost[path[-1]][i] > 0]
                if len(children) == 0:
                    path.pop()
        return []

def A_star_Traversal(cost, heuristic, start_point, goals):
    """
    Perform A* Traversal and find the optimal path
    Args:
        cost: cost matrix (list of floats/int)
        heuristic: heuristics for A* (list of floats/int)
        start_point: Staring node (int)
        goals: Goal states (list of ints)
    Returns:
        path: path to goal state obtained from A*(list of ints)
    """

    graph = Graph(cost, start_point, goals, heuristic)
    path = graph.aStar()
```

```
    # TODO
    return path

def DFS_Traversal(cost, start_point, goals):
    """
    Perform DFS Traversal and find the optimal path
        cost: cost matrix (list of floats/int)
        start_point: Staring node (int)
        goals: Goal states (list of ints)
    Returns:
        path: path to goal state obtained from DFS(list of ints)
    """

    graph = Graph(cost, start_point, goals)
    path = graph.dfs()

    # TODO
    return path
```

Output :

```
sr42@zephyrus-g14:~/Projects/MI-lab/week2$ python SampleTest.py --
SRN PES1UG20CS435
Test Case 1 for A* Traversal PASSED
Test Case 2 for DFS Traversal PASSED
sr42@zephyrus-g14:~/Projects/MI-lab/week2$
```