LAB-3
8-puzzle

final state = [0, 1, 2]
[3, 4, 5]
[6, 7, 8]

def manhattan (state, final)

1) Define the goal state in a 3×3 matrix
2) Function to find blank space

```
for i in range
    for j in range
        if state[i][j]==0
            return[i][j].
```

Once blank tile is found we move one of the four directions, up, down, left and right.

```
4 5 7        4 5 7
8 0 6   ->   8 6 0
3 1 2        3 1 2
```

this state is added to stack, again the blank space move one of the other directions, either up, down, left, right

```
4 5 0        4 5 7        4 5 7
8 6 7   ->   8 6 2   ->   8 0 6
3 1 2        3 1 0        5 1 2
```

is already visited, so move is ignored. This continues until goal state is matched and the moves are returned

Date / /
Page

Converting each ~~node~~ state into a node, after it is marked as visited the node is popped and it becomes the current ~~node~~ state,
Every valid neighbor is pushed into the stack.

```
neighbors = get_neighbors(current)

    for neighbor in neighbors:
        if neighbor not in visited
            stack.append(neighbor)

stack = [
    Node(up),
    Node(down),
    Node(left),
    Node(right)
]
```

Node (right) is popped

and it is explored

```
1 2 3        1 2 0
4 6 0   ->   4 6 3
7 5 8        7 5 8

     ->      1 2 3          1 2 3
             4 6 8    ->    4 0 6
             7 5 0          7 5 8
```

added to stack and LIFO is checked.

Date __/__/__
Page ____

```
class Node
    def _init_ (self, state, parent = None, move = None,
                                        depth = 0):
        self.state = state
        self.parent = parent
        self.move = move
        self.depth = depth

def goal_state (state)

    return state = [[1, 2, 3],
                    [4, 5, 6],
                    [7, 8, 0]]

def find_blank_tile (state):
    for i in range (len (state)):
        for j in range (len (state [i])):
            if state [i][j] = = 0:
                return (i, j)

def neighbors (node):
    state = node.state
    row, col = find_blank_tile (state)
    neighbor = []

    moves = { 'up': (row-1, col),
              'down' : (row+1, col),
              'left' : (row, co-1),
              'right' : (row, col+1),
            }
```

```
for row, (new_row, new_col) in moves.items():

        new_state[row][col]
        neighbors.append(Node(new_state, node))

def dfs_limit (start_state, depth_limit):
        stack = [Node(start_state)]
        visited = set()

    while stack:
        current_node = stack.pop()

        if is_goal(current_node.state):
            return reconstruct_path(current_node)

    visited.add(tuple(map(tuple, current_node.state)))

    if current_node.depth < depth_limit:
        neighbors = get_neighbors(current_node)
        for neighbor in neighbors:
            if (tuple(map(tuple, neighbor.state)) not in visited
                stack.append(neighbor)
        return None

def reconstruct_path(node):
    path = []
    while node.parent is not None:
        path.append(node.move)
        node = node.parent
    return path[::-1]
```

Date ___/___/___

Page ______

```
initial_state = [ [1,2,3],
                  [4,0,6],
                  [7,5,8]
                ]

depth_limit = 10
solution = dfs_limit(initial_state, depth_limit)
```

Output

Solution: ['right', 'down', 'left', 'up', 'right', 'down', 'left', 'up', 'right', 'down']

15/10