LAB - 03.

08/10/24

⇒ write an algorithm and a program to solve 8-puzzle game.

→ Algorithm to find manhattan distance.

step 1: A 2D array to represent the current state of the puzzle. (3x3)

A 2D array to represent the goal state of the puzzle (3x3).

goal state =

1	2	3
4	5	6
7	8	

= [(1, 2, 3), (4, 5, 6), (7, 8, -)]

possible moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

step 2:

```
def manhattan(state):
    for i in range(3):
        for j in range(3):
            if (state[i][j] != '-'):
                goal_i, goal_j = divmod(
                    state[i][j]-1,3)
                distance += [abs(i-goal_i)
                    -abs(j-goal-j))
            end if
        end for
    end for.
return distance.
```

step 3: checking for the goal state in current state (if currentstate == goal state)

```
def check(state):
    return goalstate == current_state
```

⇒ DFS Algorithm.

step 4:

```
def neighbours(state):
```

→ Loop the matrix from i to 3 & j to 3

→ where if state[i][j] = '-' ⇒ then check all the four directions

→ using the move matrix, we can do this

→ add those to a new array.

→ Return the neighbours array.

Step 5: DFS.

→ declare a queue

→ visited array to mark visited pos

→ loop it until the queue is empty.

→ check if the current state is the goal state.

→ skip already visited state using visited array.

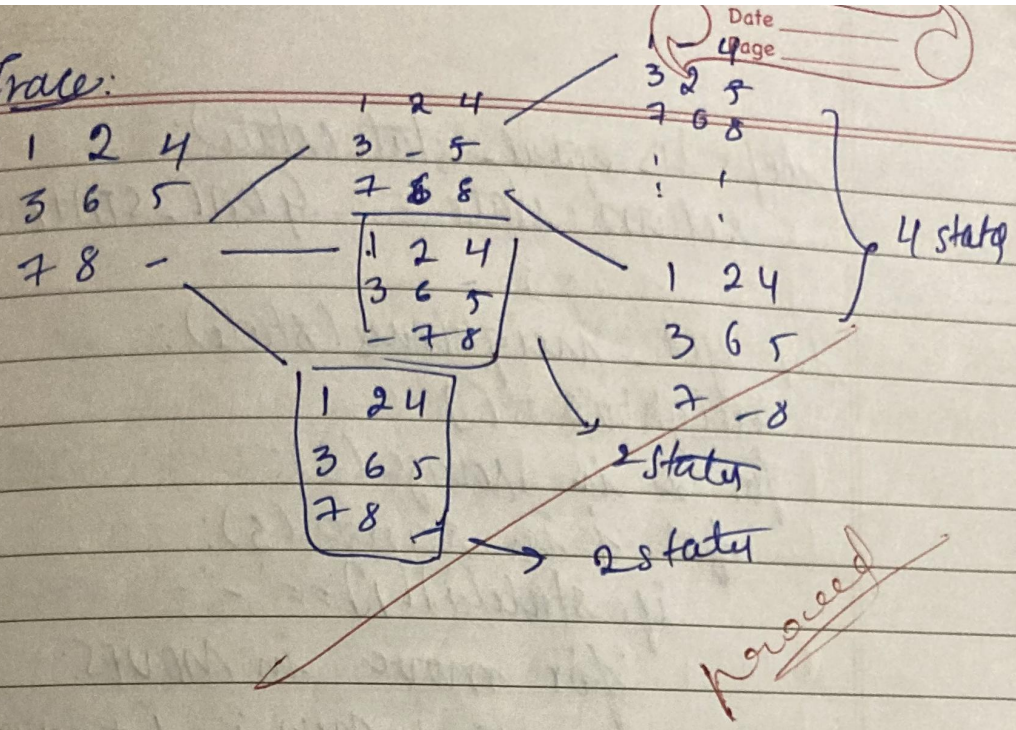
→ if not visited - add to the visited array

→ Again, get the neighbours.

if no neighbours → none.

→ end.

Trace:



⇒ code implementation:

```
from collections import deque
GOAL_STATE = [ [1, 2, 3],
               [4, 5, 6],
               [7, 8, '-'] ]
MOVES = [ (-1, 0),   # Up
          (1, 0),    # down
          (0, -1),   # Left
          (0, 1) ]   # Right.

def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != '-':
                goal_i, goal_j = divmod(state[i][j]-
                                        -1, 3)
                distance += abs(i-goal_i) + abs(j-goal_
                                                j)
    return distance.
```

```
def is_goal_state(state):
    return state == GOAL_STATE

def get_neighbours(state):
    neighbors = []
    for i in range(3):
        for j in range(3):
            if state[i][j] == '_':
                for move in MOVES:
                    new_i, new_j = i + move[0], j + move[1]
                    if 0 <= new_i < 3 and 0 <= new_j < 3:
                        new_state = [row[:] for row in state]
                        new_state[i][j], new_state
[new_i][new_j] = new_state[new_i][new_j],
new_state[i][j]

                        neighbors.append(new_state)
    return neighbors

def dfs(state):
    queue = deque([(state, [state])])
    visited = set()

    while queue:
        current_state, path = queue.popleft()

        if is_goal_state(current_state):
            return path
        if tuple(map(tuple, current_state)) in visited:
            continue
```

visited.add(tuple(map(tuple,current_state)))

for neighbor in get_neighbors(current_state):
queue.append((neighbor,path+[neighbor]))
return None.

initial_state = [[4,1,3],
[7,2,6],
[5,8,'-']]

path = dfs(initial_state)

if path:
print("Solution found:")
for state in path:
for row in state:
print(row)
print()
else:
print("No solution found.")

output:
Solution found:

[4, 1, 3]
[7, 2, 6]
[5, 8, '-']

P.T.O.

[4, 1, 3]
[7, 2, 6]
[5, '-', 8]

↓

[4, 1, 3]
[7, 2, 6]
['-', 5, 8]

↓

[4, 1, 3]
['-', 2, 6]
[7, 5, 8]

↓

['-', 1, 3]
[4, 2, 6]
[7, 5, 8]

↓

[1, '-', 3]
[4, 2, 6]
[7, 5, 8]

↓

[1, 2, 3]
[4, '-', 6]
[7, 5, 8]

↓

[1, 2, 3]
[4, 5, 6]
[7, '-', 8]

→

[1, 2, 3]
[4, 5, 6]
[7, 8, '-']

8/10/24