8 Puzzle Game

goal state =

{ [1 2 3
4 5 6
7 8 0] }

moves = { (1,0) (0,1) (-1,0) (0,-1) }
down right left up

```
def manhattan(state)
  for i in range(3)
    for j in range(3)
      if (state[i][j] != 0)
        goal_i, goal_j = divmod(state[i][j] - 1, 3)
        distance = goal_i + goal_j
```

per of elem in goal state accordingly

we check the range to 3x3 matrix

```
def getneighbours
  for i in range(3)
    for j in range(3)
      pos_i = i + move[0]
      pos_j = i + move[1]
```

This function gets the neighbouring elements once the puzzle is shuffled

This above logic gets the puzzles arranged using manhattan distance where we calculate the horizontal and vertical distance of the element in the goal state from initial state

Next in the algo function we take two variables visited and unvisited all the elements that have been checked and placed in correct order are assigned in visited list and all the elements that are not checked are placed in unvisited list we do this so that the same element is not checked in the new state again and again

1	2	3
4	5	6
7	8	0

3	1	8
7	6	4
0	2	5

curr_state = { }
goal_state = { }
stack.push(curr_state)
moves = 0

while (curr_state == goal_state)
{
left = (0, -1) → row, → column
up = (-1, 0)
right (0, 1)
down = (1, 0)

}

print (moves)

Proceed

```
from collections import deque
GOAL_STATE = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 0]
]
MOVES = [
    (-1, 0),
    (1, 0),
    (0, -1),
    (0, 1)
]

def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                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_neighbors(state):
    neighbors = []
    for i in range(3):
        for j in range(3):
            if (state[i][j] == 0):
                for move in MOVES:
                    ni, nj = i + move[0], j + move[1]
                    if 0 <= ni < 3 and 0 <= nj < 3:
                        newstate = [row[:] for row in state]
new_state[i][j], newstate[ni][nj] = new_state[ni][nj], new_state[i][j]
neighbors.append(newstate)
    return neighbors
```

```
def dfs(state):
    queue = deque([(state, [state])])
    visited = set()
    while queue:
        cs, ph = queue.popleft()
        if is_goal_state(cs):
            return ph
        if tuple(map(tuple, cs)) in visited:
            continue
        visited.add(tuple(map(tuple, cs)))
        for neighbor in get_neighbors(cs):
            queue.append((neighbor, path + [neighbor]))
    return None
initial_state = [
    [4,1,3],
    [7,2,6],
    [5,8,0]
]
path = dfs(initial_state)
if ph:
    print("solution found")
    for state in ph:
        for row in state:
            print(row)
        print()
else:
    print("No solution found")
```

Output

solution found:

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

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

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

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

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

4	1	3
7	2	6
5	8	0

1	2	3
4	5	6
7	8	0

1+1+0+1+3+0+0+2 = 8

total moves = 8

```
Solution found:
[4, 1, 3]
[7, 2, 6]
[5, 8, 0]

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

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

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

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

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

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

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

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