PAGE : 8/10/24
DATE : / /

8-puzzle

Algorithm:

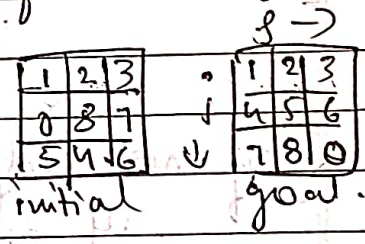
Step 1: Create goal configuration of 3×3 matrix where '0' represents the blank tile.

Step 2: defines the possible directions for moving blank tile up, down, left & right.

Step 3: Manhattan function calculates the distance which estimates the number of moves required to reach the goal

Step 4: Create function called is_goal_state which checks whether a given state matches the goal_state

step 5: In neighboring state :- It iterates through the puzzle to locate the position of '0'



Algorithm (using DFS)

```
start_state = [...]
goal_state = [...]
stack.push(start_state)
visited_set = { }
moves = 0
f(i,j)
{
  visited_set.add(current state)
  if(current state == goal_state)
  if(not in visited_set)
  {
      left = f(i, j+1)
      right(i, j-1)
      up = f(i-1, j)
      down = f(i+1, j)
  }
}
print(moves)
```

PAGE :
DATE : / /

```
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):
    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:
                    new_i, new_j = i + move[0], j + move[1]
                    if 0 <= new_i < 3 and 0 <= new_j < 3:
```

PAGE :
DATE : / /

```
        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, 0]
]
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.")
```

PAGE :
DATE : / /

o/p: Solution found

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

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

Total moves = 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]

```
[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]


=== Code Execution Successful ===
```

```
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]
```