DATE: PAGE:

1	2	3
4	0	5
6	7	8

* 8 Puzzle Game

- Initialize a goal_state = [[1,2,3], [4,5,6], [7,8,0]]
- Randomly generate a current initial state
- Run till cur_state != goal_state
  calculate manhattan distance
  if m_dis = 0 break
  else compare (cur_state, goal_state)
  select the min manhattan and move
  initialize cur_state to that
  repeatedly.

check if cur_state == goal_state
v. break and display no of moves.

else
Run until cur_state != goal_state
- calculate manhattan distance
- compare (cur_state, goal_state) and select the minimum distance out of all available states wrt to goal_state.
- Initialize to cur_state.

DATE: PAGE:

Using DFS.

Initialize goal state, all_possible_moves for center, side, corner.

Set limit = n

- Calculate available moves for that index
- Randomly allocate a move if it is at center, (has 4 legal moves)
- if it is at side, it has 3 possible moves
- if corner, (2 possible moves)

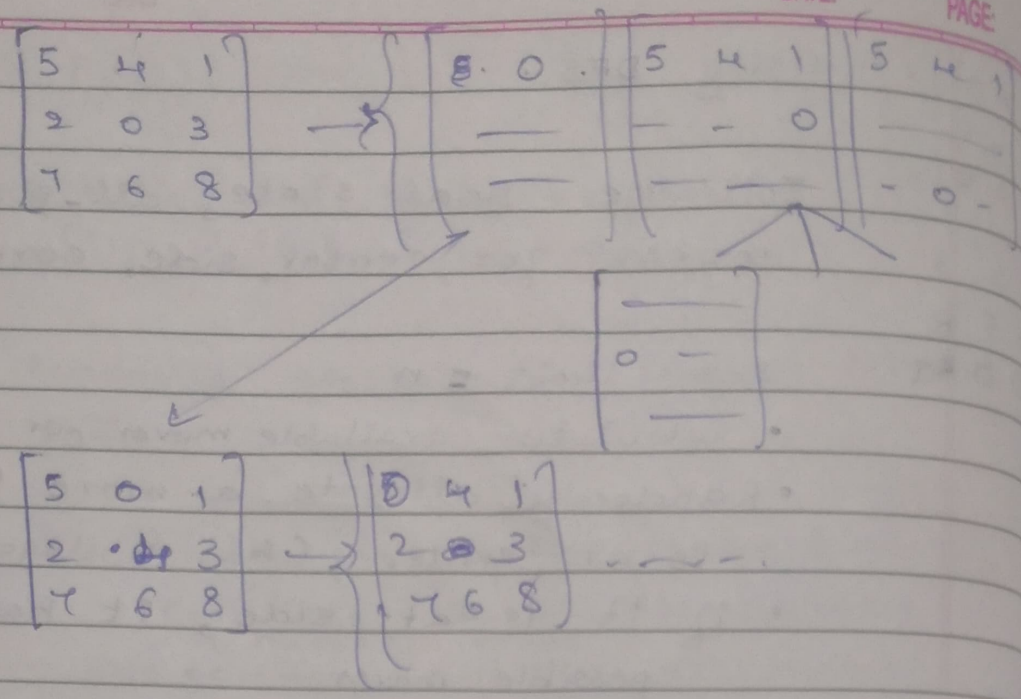
- Make it as visited

- After going through the depth at any step if it gets equal to goal_state start returning by printing sequence.

- if limit is reached go back to previous step and go to other available node.

Repeat recursively

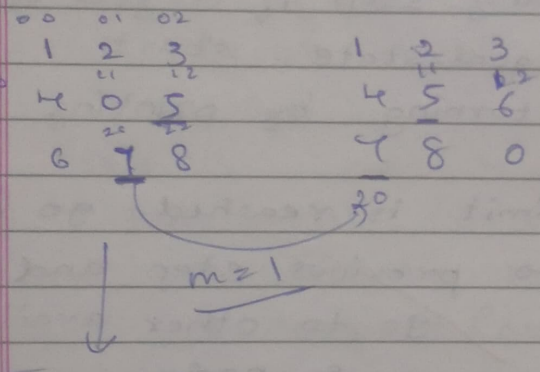
DATE: PAGE:

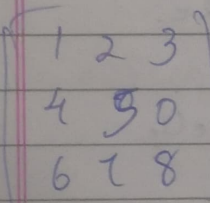


If limit is reached and starts next available

Manhattan distance.

GrS





DATE: PAGE:

```
from copy import deepcopy

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

def find_blank_tile(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
    return None

def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            tile = state[i][j]
            if tile != 0:
                goal_x, goal_y = divmod
                                 ((t-1), 3)
                distance += abs(goal_x - i)
                            + abs(goal_y - j)
    return distance

def make_move(state, move):
    new_state = deepcopy(state)
    blank_x, blank_y = find_blank_tile
                       (state)
    if move == "up" and blank_x > 0:
        new_state[blank_x][blank_y],
        new_state[blank_x-1][blank_y]
        = new_state[blank_x-1][blank_y],
        new_state[blank_x][blank_y]
```

DATE: PAGE:

```
elif move == "down" and blank_x < 2:
    new_state[blank_x][blank_y], newstate
    [blockx + 1][block_y] = new_state
    [block_x+1][block_y], new_state
    [block_x][block_y]

elif move == "left" and blank_y > 0:
    new_state[blank_x][blank_y], new_sta
    [blank_x][blank_y - 1] = new_state[
    [blank_x][blank_y -1]

elif move == "right" and blank_y < 2:
    new_state[blank_x][blank_y] new_sta[blank_
    [blank_y + 1] = new_state[blank_x
                    [blank_y + 1],
                new_state[block_x][block_y]

return new_state


def get_valid_moves(state):
    blank_x, blank_y = find_blank_tile
                       (state)

    moves = []
    if blank_x > 0:
        moves.append("up")
    if blank_x < 2:
        moves.append("down")
    if blank_y > 0:
        moves.append("left")
    if blank_y < 2:
        moves.append("right")
```

DATE: PAGE:

```
return moves

def dfs(initial state):
    stack = [(initial_state, [ ])]

    visited = set()

    while stack:
        current_state = stack.pop()

        state_tuple = tuple(tuple(row)
                            for row in current_state)

        if state_tuple in visited:
            continue
        visited.add(state_tuple)

        if current_state == goal_state:
            return path

        valid_moves = get_valid_moves
                      (current_state)

        for move in valid_moves:
            new_state = make_move(curr,
                                  move)

            new_path = path + [move]
            stack.append((new_state,
                          new path))
```

DATE: PAGE:

```
    return None

if __name__ == "__main__":
    initial_state = [[1,2,3],[4,0,5],[6,7,8]]

    solution_moves = dfs_solve_puzzle(
                        initial state)

    if solution_moves:
        print("Solution found")
        print("Moves: " + " → ".join(solution_
                                    moves))
    else:
        print("No solution exists for this
                puzzle")
```

Output

Solution found

Moves: right → down → left