08/10

8 Puzzle

Algorithm: (Using manhattan distance)

Step 1: Define the goal state as $\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$

Step 2: Take the shuffled input from the user.

Step 3: Define a function to calculate the manhattan distance of the current state
Manhattan distance = $abs(x_1 - x_2) + abs(y_1 - y_2)$
x_1 = current row position
y_1 = current column position
x_2 = goal row position
y_2 = goal column position.

Step 4: Write a function to calculate the number of moves that are possible for the current state. If the blank is in the center, there are 4 possible moves. If the blank is in the edge then 3 possible moves. If the blank is in the corner then 2 possible moves.
Push all of the moves in the ~~array~~ priority queue
~~Sort the array and return~~ it. Return priority queue.

Step 5: Write a dfs function which calls itself for each possible move returned by the number of moves function.
If the current state is equal to the goal state, ~~return the~~ go to step 6
Else recursively call the dfs function
Maintain a visited map to not repeat state

Step 6: Backtrack and print the sequence of moves.

Algorithm: (Using only DFS)

Step 1: Define the goal state as $\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$

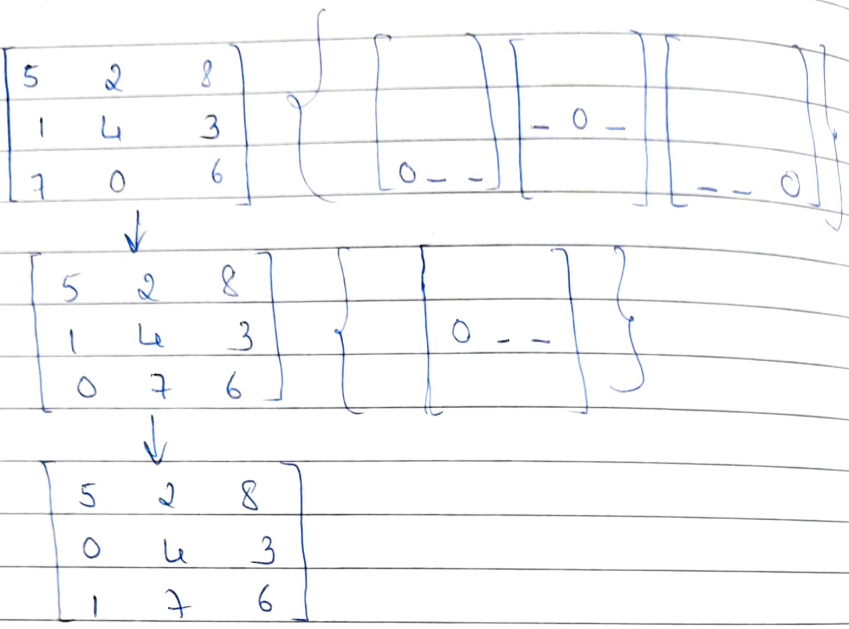
Step 2: Take the shuffled input from the user

Step 3: Write a function to calculate all possible moves. Push them in an array and return the array.

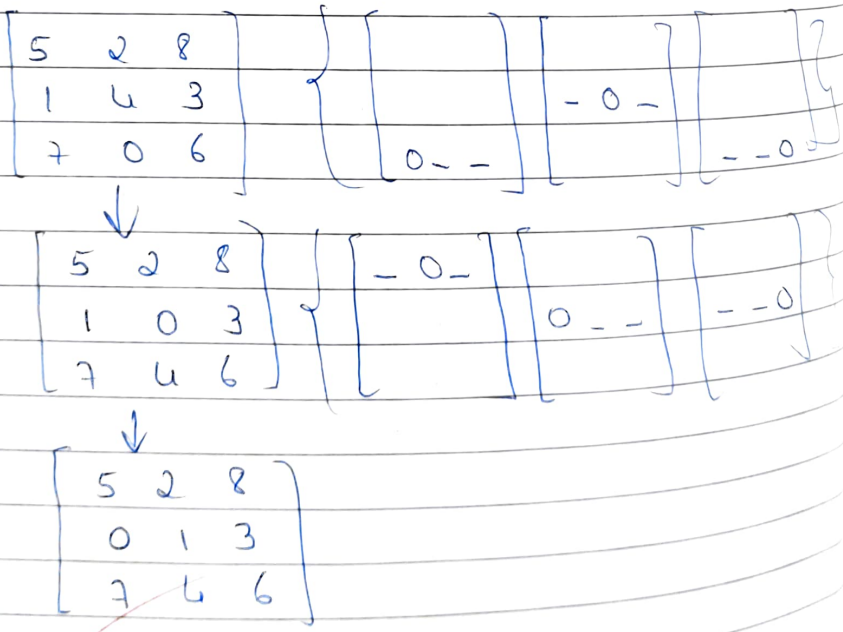
Step 4: Write a dfs function which calls itself for each possible moves returned by the moves function. If the current state is equal to the goal state, go to step 5. Else recursively call the dfs function. Maintain a visited array to not repeat states. Keep a max depth.

Step 5: Backtrack and print the sequence of moves.

DFS: Manhattan:



Manhattan DFS:



Code:

```
import heapq
import numpy as np

goal = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
vis = set()
q = []
parent_map = {}
move_map = {}

def manhattan(curr):
    ans = 0
    pos = {goal[i][j] : (i, j) for i in range(3) for j in range(3)}
    for i in range(3):
        for j in range(3):
            x, y = pos[curr[i][j]]
            ans += abs(i-x) + abs(j-y)
    return ans

def moves(curr):
    x, y = [(i, j) for i in range(3) for j in range(3) if curr[i][j] == 0][0]
    poss = [[0, -1, 'left'], [-1, 0, 'right'], [1, 0, 'down'], [0, 1, 'up']]
    for dx, dy, direction in poss:
        nx, ny = x+dx, y+dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            curr1 = [row[:] for row in curr]
            curr1[x][y], curr1[nx][ny] = curr1[nx][ny], curr1[x][y]
            tuple_curr1 = tuple(map(tuple, curr))
            if tuple_curr1 not in vis:
                heapq.heappush(q, (manhattan(curr), curr))
                vis.add(tuple_curr1)
                parent_map[tuple(map(tuple, curr))] = curr
```

```
        move_map[tuple(map(tuple,curr))] = direction

def dfs(curr):
    vis.add(tuple(map(tuple,curr)))
    if curr == goal:
        return True
    g=moves(curr)
    if g:
        curr=heapq.heappush(q)[1]
        if dfs(curr):
            return True
    return False

def display(board):
    print("+--+--+--+")
    for row in board:
        print("|"+"|".join(str(x) if x!=0
        else 'x' for x in row)+"|")
        print("+--+--+--+")

c=[[] for i in range(3)]
for i in range(3):
    print(f"Enter elements of row {i+1}")
    c[i]=list(map(int,input().split()))
dfs(c)
result_path=[]
directions=[]
state=goal

while state:
    result_path.append(state)
    directions.append(move_map.get(tuple(map(
    state=parent_map.get(tuple(map(
```

print(f"Steps taken: {len(result_path)-1}")

Output:

Enter elements: 7 6 5
3 8 4
1 0 2

Step 0:

7	6	5
3	8	4
1	0	2

:

Step 32:

0	1	2
3	4	5
6	7	8