# PROGRAM TITLE 1

# 8-PUZZLE PROBLEM

## AIM:

To write a python program to solve 8-puzzle problem.

## PROCEDURE:

1. **PuzzleNode Class:**
    - Represents a node in the search space.
    - Contains the current state of the puzzle, a reference to the parent node, the move made to reach this state, the cost of reaching this state from the initial state, and the heuristic value.
2. **Manhattan Distance Heuristic:**
    - The Manhattan distance is used as a heuristic to estimate the cost of reaching the goal state from the current state.
    - It calculates the sum of the horizontal and vertical distances between the current positions of each tile and their goal positions.
3. **get_successors Function:**
    - Generates successor nodes by exploring possible moves (up, down, left, right) from the current state.
    - Ensures that successor states are valid and not previously visited.
4. **solve_8_puzzle Function:**
    - Implements the A* search algorithm to find the solution.
    - Uses a priority queue to explore nodes with lower estimated costs first.
    - Continues until the goal state is reached or the search space is exhausted.
5. **find_empty_tile Function:**
    - Helper function to find the position of the empty tile (0) in the puzzle state.
6. **get_solution_path Function:**

- Constructs the solution path by backtracking from the goal node to the initial node.

7. **Print State Function:**

   - Helper function to print the state of the puzzle.

## CODING:

```
import copy

from heapq import heappush, heappop


n = 3


row = [1, 0, -1, 0] col

= [0, -1, 0, 1]



class priorityQueue:


    def __init__(self):
self.heap = []


    def push(self, k):
        heappush(self.heap, k)


    def pop(self):
        return heappop(self.heap)
```

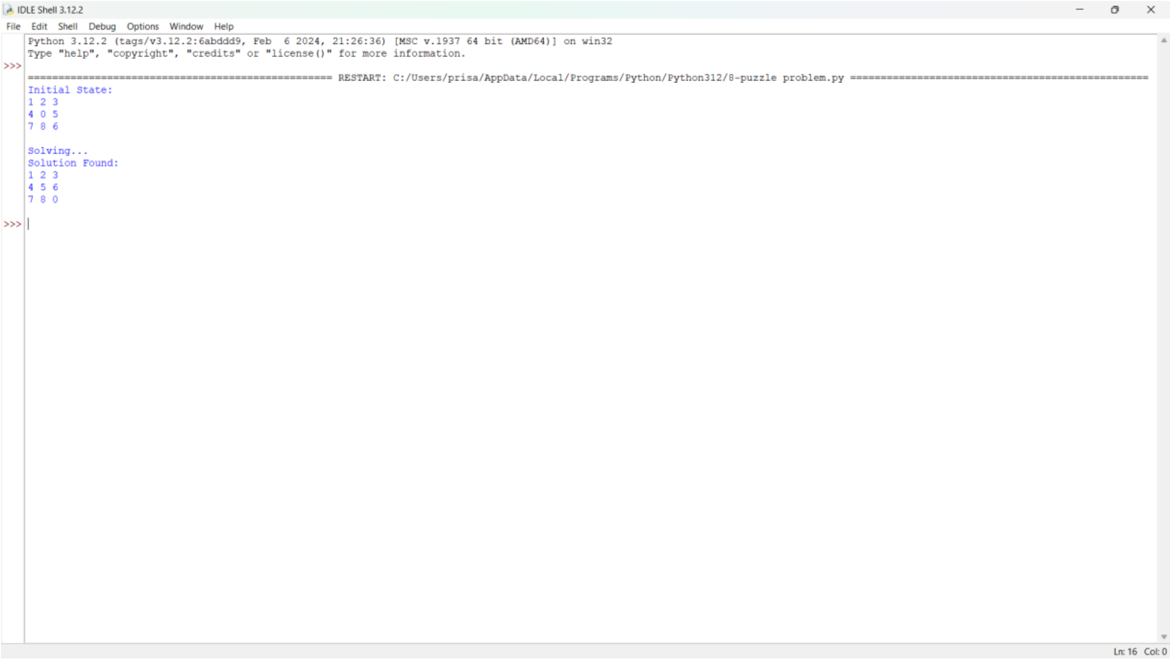
```
    def     empty(self):
if     not     self.heap:
return True         else:
            return False



class node:


    def __init__(self, parent, mat, empty_tile_pos,
                 cost, level):
self.parent = parent


        self.mat = mat


        self.empty_tile_pos = empty_tile_pos


        self.cost = cost


        self.level = level


    def __lt__(self, nxt):
        return self.cost < nxt.cost



def calculateCost(mat, final) -> int:
    count = 0      for i in
range(n):          for j in
range(n):              if
((mat[i][j]) and
```

```
                (mat[i][j] != final[i][j])):
            count += 1

    return count


def newNode(mat, empty_tile_pos, new_empty_tile_pos,
          level, parent, final) -> node:
    new_mat = copy.deepcopy(mat)

    x1 = empty_tile_pos[0]     y1 = empty_tile_pos[1]     x2 =
new_empty_tile_pos[0]     y2 = new_empty_tile_pos[1]     new_mat[x1][y1],
new_mat[x2][y2] = new_mat[x2][y2], new_mat[x1][y1]

    cost = calculateCost(new_mat, final)

    new_node = node(parent, new_mat, new_empty_tile_pos,
                cost, level)
    return new_node


def printMatrix(mat):
for i in range(n):
for j in range(n):

          print("%d " % (mat[i][j]), end=" ")

print()
```

```
def isSafe(x, y):     return x >= 0 and x < n and
y >= 0 and y < n



def printPath(root):
if root == None:
        return

    printPath(root.parent)
printMatrix(root.mat)     print()



def solve(initial, empty_tile_pos, final):
pq = priorityQueue()

    cost = calculateCost(initial, final)
root = node(None, initial,
empty_tile_pos, cost, 0)

    pq.push(root)

    while not pq.empty():

        minimum = pq.pop()

        if minimum.cost == 0:
printPath(minimum)
```

```
            return

        for i in range(4):              new_tile_pos = [
minimum.empty_tile_pos[0] + row[i],
minimum.empty_tile_pos[1] + col[i], ]

            if isSafe(new_tile_pos[0], new_tile_pos[1]):
child = newNode(minimum.mat,
minimum.empty_tile_pos,
new_tile_pos,                                  minimum.level +
1,                                   minimum, final, )

                pq.push(child)



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

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

empty_tile_pos = [1, 2]

solve(initial, empty_tile_pos, final)
```

## OUTPUT:




## RESULT:

Hence the program been successfully executed and verified.