# 8-Puzzle Problem Solver Using A\* Algorithm (Python)

## Python Code:

import heapq  
  
GOAL\_STATE = [[1,2,3],[4,5,6],[7,8,0]]  
DIRECTIONS = [(-1,0), (1,0), (0,-1), (0,1)]  
  
class PuzzleNode:  
 def \_\_init\_\_(self, state, parent=None, move="", depth=0, cost=0):  
 self.state = state  
 self.parent = parent  
 self.move = move  
 self.depth = depth  
 self.cost = cost  
  
 def \_\_lt\_\_(self, other):  
 return self.cost < other.cost  
  
def is\_goal(state):  
 return state == GOAL\_STATE  
  
def find\_blank(state):  
 for i in range(3):  
 for j in range(3):  
 if state[i][j] == 0:  
 return i, j  
  
def manhattan\_distance(state):  
 distance = 0  
 for i in range(3):  
 for j in range(3):  
 val = state[i][j]  
 if val != 0:  
 goal\_x = (val - 1) // 3  
 goal\_y = (val - 1) % 3  
 distance += abs(goal\_x - i) + abs(goal\_y - j)  
 return distance  
  
def get\_neighbors(node):  
 neighbors = []  
 x, y = find\_blank(node.state)  
 for dx, dy in DIRECTIONS:  
 nx, ny = x + dx, y + dy  
 if 0 <= nx < 3 and 0 <= ny < 3:  
 new\_state = [row[:] for row in node.state]  
 new\_state[x][y], new\_state[nx][ny] = new\_state[nx][ny], new\_state[x][y]  
 move = f"({x},{y}) → ({nx},{ny})"  
 neighbor\_node = PuzzleNode(  
 state=new\_state,  
 parent=node,  
 move=move,  
 depth=node.depth + 1,  
 cost=node.depth + 1 + manhattan\_distance(new\_state)  
 )  
 neighbors.append(neighbor\_node)  
 return neighbors  
  
def state\_to\_tuple(state):  
 return tuple(tuple(row) for row in state)  
  
def a\_star\_search(start\_state):  
 start\_node = PuzzleNode(state=start\_state, cost=manhattan\_distance(start\_state))  
 open\_set = []  
 heapq.heappush(open\_set, start\_node)  
 visited = set()  
  
 while open\_set:  
 current = heapq.heappop(open\_set)  
 if is\_goal(current.state):  
 return reconstruct\_path(current)  
 visited.add(state\_to\_tuple(current.state))  
 for neighbor in get\_neighbors(current):  
 if state\_to\_tuple(neighbor.state) not in visited:  
 heapq.heappush(open\_set, neighbor)  
 return None  
  
def reconstruct\_path(node):  
 path = []  
 while node.parent:  
 path.append((node.move, node.state))  
 node = node.parent  
 path.reverse()  
 return path  
  
def print\_state(state):  
 for row in state:  
 print(" ".join(str(num) if num != 0 else "\_" for num in row))  
 print()  
  
def main():  
 start\_state = [[1, 2, 3], [4, 0, 6], [7, 5, 8]]  
 print("Initial State:")  
 print\_state(start\_state)  
 path = a\_star\_search(start\_state)  
 if path:  
 print(f"Solved in {len(path)} moves:\n")  
 for i, (move, state) in enumerate(path):  
 print(f"Move {i+1}: {move}")  
 print\_state(state)  
 else:  
 print("No solution found.")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

## Sample Output:

Initial State:  
1 2 3  
4 \_ 6  
7 5 8  
  
Solved in 2 moves:  
  
Move 1: (2,1) → (1,1)  
1 2 3  
4 5 6  
7 \_ 8  
  
Move 2: (2,2) → (2,1)  
1 2 3  
4 5 6  
7 8 \_