A\* algorithm (8-puzzle problem):

import heapq

class PuzzleState:

    def \_\_init\_\_(self, board, move\_count, prev\_state):

        self.board = board

        self.move\_count = move\_count

        self.prev\_state = prev\_state

        self.zero\_pos = self.find\_zero()

    def find\_zero(self):

        for i in range(len(self.board)):

            if self.board[i] == 0:

                return i

    def generate\_neighbors(self):

        neighbors = []

        row, col = divmod(self.zero\_pos, 3)

        possible\_moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

        for dr, dc in possible\_moves:

            new\_row, new\_col = row + dr, col + dc

            if 0 <= new\_row < 3 and 0 <= new\_col < 3:

                new\_pos = new\_row \* 3 + new\_col

                new\_board = self.board[:]

                new\_board[self.zero\_pos], new\_board[new\_pos] = new\_board[new\_pos], new\_board[self.zero\_pos]

                neighbors.append(PuzzleState(new\_board, self.move\_count + 1, self))

        return neighbors

    def manhattan\_distance(self, goal):

        distance = 0

        for i in range(1, 9):

            current\_pos = self.board.index(i)

            goal\_pos = goal.index(i)

            current\_row, current\_col = divmod(current\_pos, 3)

            goal\_row, goal\_col = divmod(goal\_pos, 3)

            distance += abs(current\_row - goal\_row) + abs(current\_col - goal\_col)

        return distance

    def \_\_lt\_\_(self, other):

        return (self.move\_count + self.manhattan\_distance(goal\_state)) < (other.move\_count + other.manhattan\_distance(goal\_state))

def a\_star(initial, goal):

    open\_list = []

    closed\_set = set()

    initial\_state = PuzzleState(initial, 0, None)

    heapq.heappush(open\_list, initial\_state)

    while open\_list:

        current\_state = heapq.heappop(open\_list)

        if current\_state.board == goal:

            return current\_state

        closed\_set.add(tuple(current\_state.board))

        for neighbor in current\_state.generate\_neighbors():

            if tuple(neighbor.board) not in closed\_set:

                heapq.heappush(open\_list, neighbor)

    return None

def print\_solution(solution):

    path = []

    while solution:

        path.append(solution.board)

        solution = solution.prev\_state

    path.reverse()

    for step in path:

        print\_board(step)

        print()

def print\_board(board):

    for i in range(3):

        print(board[3\*i : 3\*(i+1)])

initial\_state = [

    1, 2, 3,

    0, 4, 6,

    7, 5, 8

]

goal\_state = [

    1, 2, 3,

    4, 5, 6,

    7, 8, 0

]

solution = a\_star(initial\_state, goal\_state)

if solution:

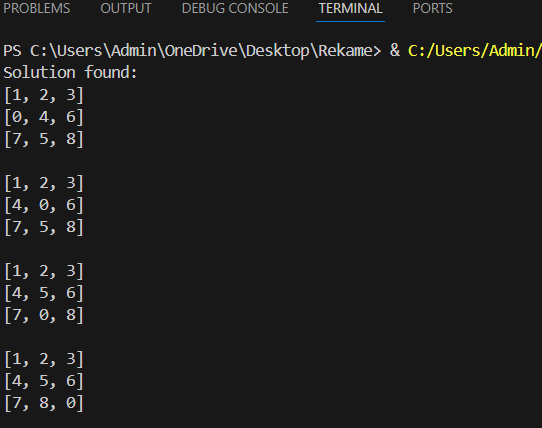
    print("Solution found:")

    print\_solution(solution)

else:

    print("No solution found.")

**Output:**

****