8-Puzzle Solver

Name: Shriyans Jain

Roll No: 2024011100300242

Course: Introduction To AI

Branch: CSE A

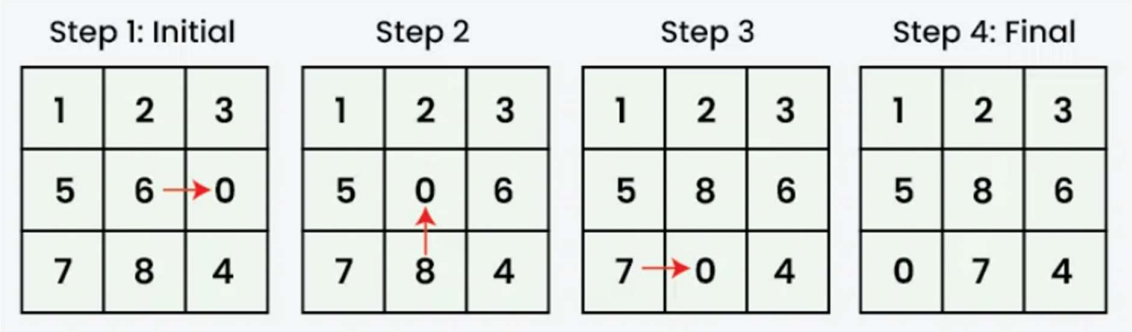
Section: D

Date: 11/03/2025

INTRODUCTION:

PROBLEM STATEMENT:

The task is to develop an intelligent solver for the 8-puzzle problem, a sliding tile puzzle that consists of a 3×3 grid containing eight numbered tiles (1-8) and a blank space (0). The goal of the AI rearrange the tiles into the correct order. The problem is solved using A* search, where the input of the tiles are given by human.



User Input Handling: The program prompts the user to enter the puzzle's initial state while ensuring valid input.

*A Search Algorithm:** It finds the optimal path to the solution by prioritizing states with the lowest cost (depth + heuristic).

Manhattan Distance Heuristic: This heuristic estimates how far tiles are from their correct positions, guiding the search efficiently.

Priority Queue (Min-Heap): Ensures that the most promising states are explored first.

Solution Path Display: The program prints each move taken and the depth at which the solution is reached

METHODOLOGY:

The 8-puzzle problem is a classic state-space search problem that involves arranging tiles in a 3×3 grid to match a predefined goal configuration. The solution must determine a sequence of moves required to reach the goal from a user-defined initial state. To achieve this, **Depth-First Search (DFS)** is implemented to explore possible moves systematically.

Problem Formulation

The puzzle consists of eight numbered tiles (1-8) and a blank space (0), which can move **left, right, up, or down** within the grid. The objective is to transform a given initial state into the goal state:

Algorithm Selection: Depth-First Search (DFS)

DFS is a blind search algorithm that explores a path as deep as possible before backtracking. It does not guarantee the shortest path but is simple and effective for small problem spaces. The algorithm is implemented using a stack (LIFO) and follows these steps:

User Input Handling – The program verifies that the input configuration is valid and ensures that numbers 0-8 are entered correctly.

State Representation – A PuzzleState class is used to store the board configuration, depth, blank tile position, and solution path.

DFS Execution

The initial state is pushed onto the stack.

The algorithm explores possible moves (up, down, left, right) and generates new board states.

If a state has not been visited, it is added to the stack for further exploration.

The search continues until the goal state is reached or all possibilities are exhausted.

1. *Solution Path Display – The sequence of moves taken to reach the goal is printed*

CODE:

```python
# Import necessary libraries
from collections import deque

# Define the dimensions of the puzzle
N = 3

# Structure to store a state of the puzzle
class PuzzleState:
    def __init__(self, board, x, y, depth, path):
        self.board = board
        self.x = x
        self.y = y
        self.depth = depth
        self.path = path

# Possible moves: Left, Right, Up, Down
row_moves = [0, 0, -1, 1]
col_moves = [-1, 1, 0, 0]

# Function to check if a given state is the goal state
def is_goal_state(board):
    goal = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
    return board == goal

# Function to check if a move is valid
def is_valid(x, y):
    return 0 <= x < N and 0 <= y < N

# Function to print the board
def print_board(board):
    for row in board:
```

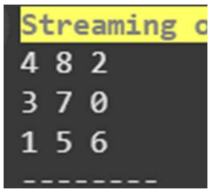
```
        print(' '.join(map(str, row)))
    print("--------")

# Function to take user input for the board with validation
def get_user_input():
    print("Enter the 8-puzzle numbers row by row (use 0 for the blank space):")
    board = []
    numbers = set()

    for i in range(N):
        while True:
            try:
                row = list(map(int, input(f"Row {i+1}: ").split()))
                if len(row) != N:
                    raise ValueError("Each row must contain exactly 3 numbers.")
                if any(num < 0 or num > 8 for num in row):
                    raise ValueError("Numbers must be between 0 and 8.")
                if any(num in numbers for num in row):
                    raise ValueError("Duplicate numbers are not allowed.")
                numbers.update(row)
                break
            except ValueError as e:
                print(f"Invalid input: {e}. Please try again.")
        board.append(row)

    # Ensure there's exactly one 0 (blank space)
```

```
    # Ensure there's exactly one 0 (blank space)
    if 0 not in numbers:
        print("Error: The input must contain exactly one 0 for the blank space.")
        return get_user_input()

    # Find the position of the blank space (0)
    x, y = next((i, j) for i in range(N) for j in range(N) if board[i][j] == 0)
    return board, x, y

# Depth-First Search (DFS) to solve the 8-puzzle problem
def solve_puzzle_dfs(start, x, y):
    stack = []  # Stack for DFS
    visited = set()
    stack.append(PuzzleState(start, x, y, 0, []))
    visited.add(tuple(map(tuple, start)))

    while stack:
        curr = stack.pop()

        # Print the current board
        print(f'Depth: {curr.depth}')
        print_board(curr.board)

        # Check if goal state is reached
        if is_goal_state(curr.board):
            print(f'Goal state reached at depth {curr.depth}')
            print("Solution Path:")
            for step in curr.path:
                print_board(step)
            return
```

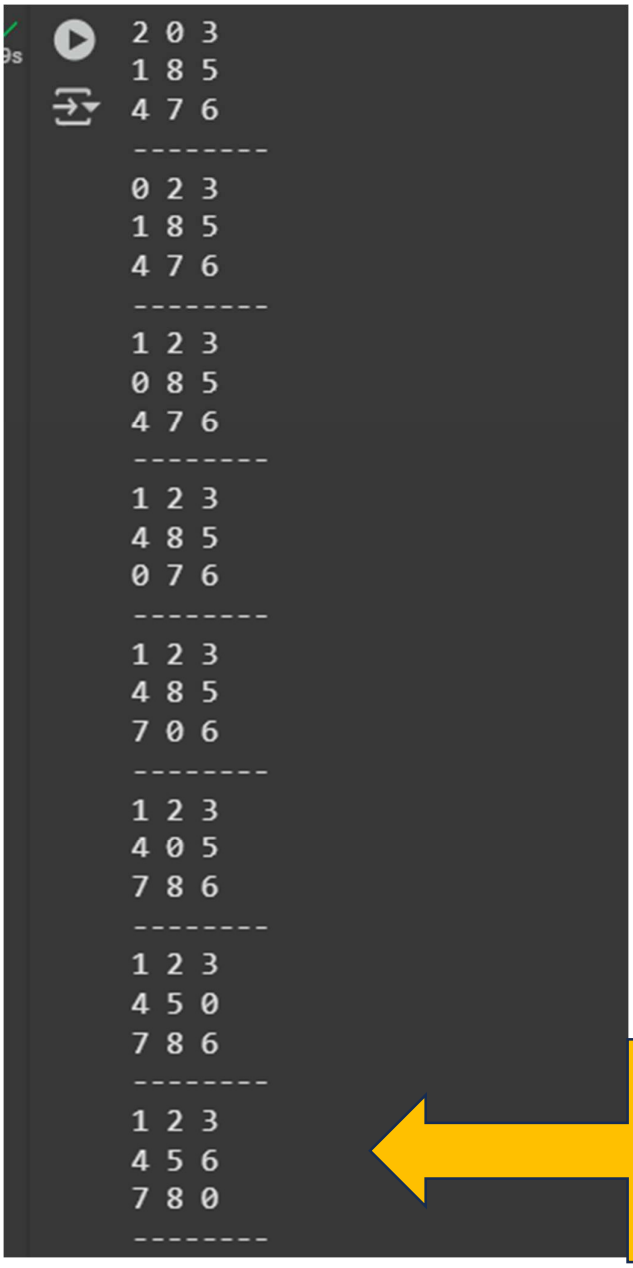
```
            return

        # Explore possible moves
        for i in range(4):
            new_x = curr.x + row_moves[i]
            new_y = curr.y + col_moves[i]

            if is_valid(new_x, new_y):
                new_board = [row[:] for row in curr.board]
                # Swap the tiles
                new_board[curr.x][curr.y], new_board[new_x][new_y] = new_board[new_x][new_y], new_board[curr.x][curr.y]

                # If this state has not been visited before, push to stack
                board_tuple = tuple(map(tuple, new_board))
                if board_tuple not in visited:
                    visited.add(board_tuple)
                    stack.append(PuzzleState(new_board, new_x, new_y, curr.depth + 1, curr.path + [new_board]))

    print('No solution found (DFS reached depth limit)')

# Driver Code
if __name__ == '__main__':
    start, x, y = get_user_input()

    print('Initial State:')
    print_board(start)

    solve_puzzle_dfs(start, x, y)
```

OUTPUT/RESULT:

INITAIL INPUT:



OUTPUTS and proccess:



THIS IS THE FINAL OUTPUT, THE NUMBER TILES ARE ARRANGED IN INCREASING ORDER PER ROW

REFERNCE AND CREDITS ;):

Depth-First Search (DFS)- For understanding the Depth-First Search (DFS) and 8 tile puzzle solving, I referred to various online resources and tutorials:

GEEKSFORGEEKS:

8 puzzle Problem - GeeksforGeeks

WIKEPEDIA:

Depth-first search - Wikipedia