8 Puzzle Solver

Introduction to AI

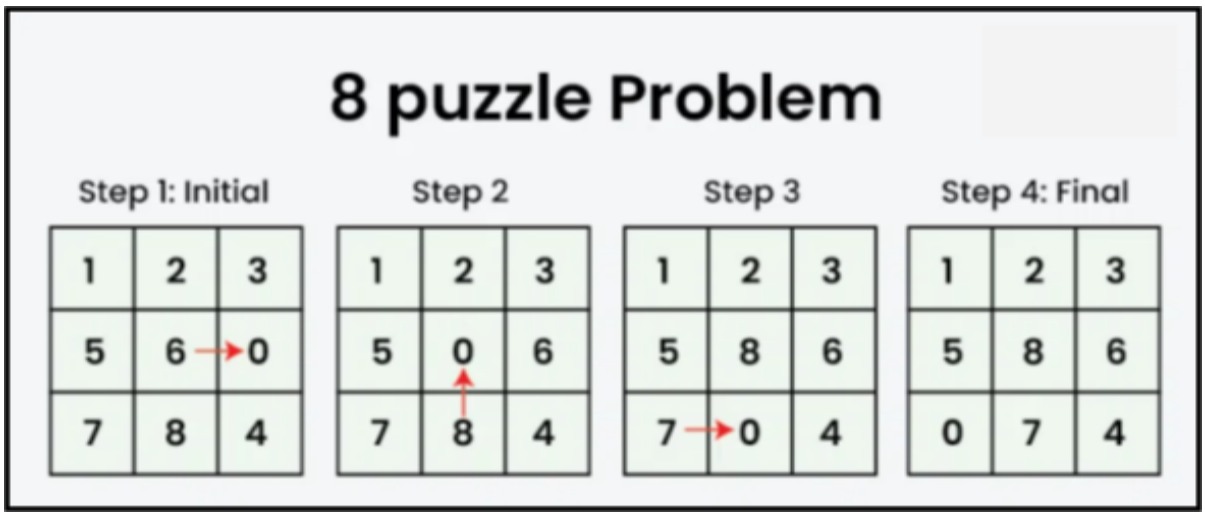
MSE - 1

By Krish Sandhu

BTech CSE (AI) - 2024011100300138

Introduction to 8 Puzzle

The 8 puzzle is a 3×3 board with 8 tiles (each numbered from 1 to 8) and one empty space, the objective is to place the numbers to match the final configuration (numbers sorted in ascending order) using the empty space. We can slide four adjacent tiles (left, right, above, and below) into the empty space.



Methodology

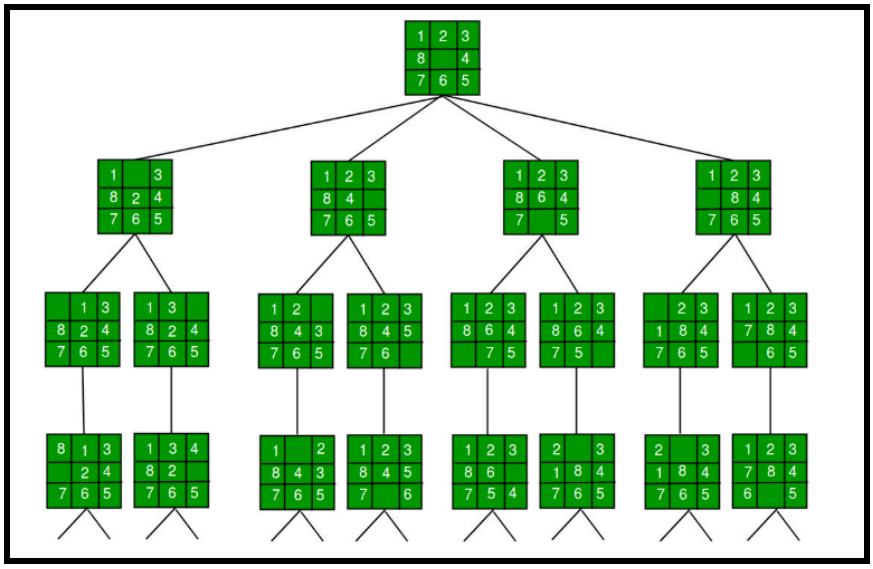
To solve this problem I will utilize a Breadth-first search (BFS) algorithm on the state space tree. This always finds a goal state nearest to the root.

Using BFS – O(n!) Time and O(n!) Space

- Breadth-first search on the state-space tree.
- Always finds the nearest goal state.
- Same sequence of moves irrespective of initial state.

Step by step approach

- Start from the root node.
- Explore the leftmost child node recursively until you reach a leaf node or a goal state.
- If a goal state is reached, return the solution.
- If a leaf node is reached without finding a solution, backtrack to explore other branches.



Code Typed

```
# Import necessary libraries
from collections import deque

# Define the dimensions of the puzzle
N = 3

# Class to represent the state of the puzzle
class PuzzleState:
    def __init__(self, board, x, y, depth):
        self.board = board
        self.x = x
        self.y = y
        self.depth = depth

# Possible moves: Left, Right, Up, Down
row = [0, 0, -1, 1]
col = [-1, 1, 0, 0]

# Function to check if the current 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 puzzle board
def print_board(board):
    for row in board:
        print(' '.join(map(str, row)))
    print('--------')
```

```
# BFS function to solve the 8-puzzle problem
def solve_puzzle_bfs(start, x, y):
    q = deque()
    visited = set()

    # Enqueue initial state
    q.append(PuzzleState(start, x, y, 0))
    visited.add(tuple(map(tuple, start)))

    while q:
        curr = q.popleft()

        # Print the current board state
        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}')
            return

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

            if is_valid(new_x, new_y):
                new_board = [row[:] for row in curr.board]
                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 queue
                if tuple(map(tuple, new_board)) not in visited:
                    visited.add(tuple(map(tuple, new_board)))
                    q.append(PuzzleState(new_board, new_x, new_y,
curr.depth + 1))
```

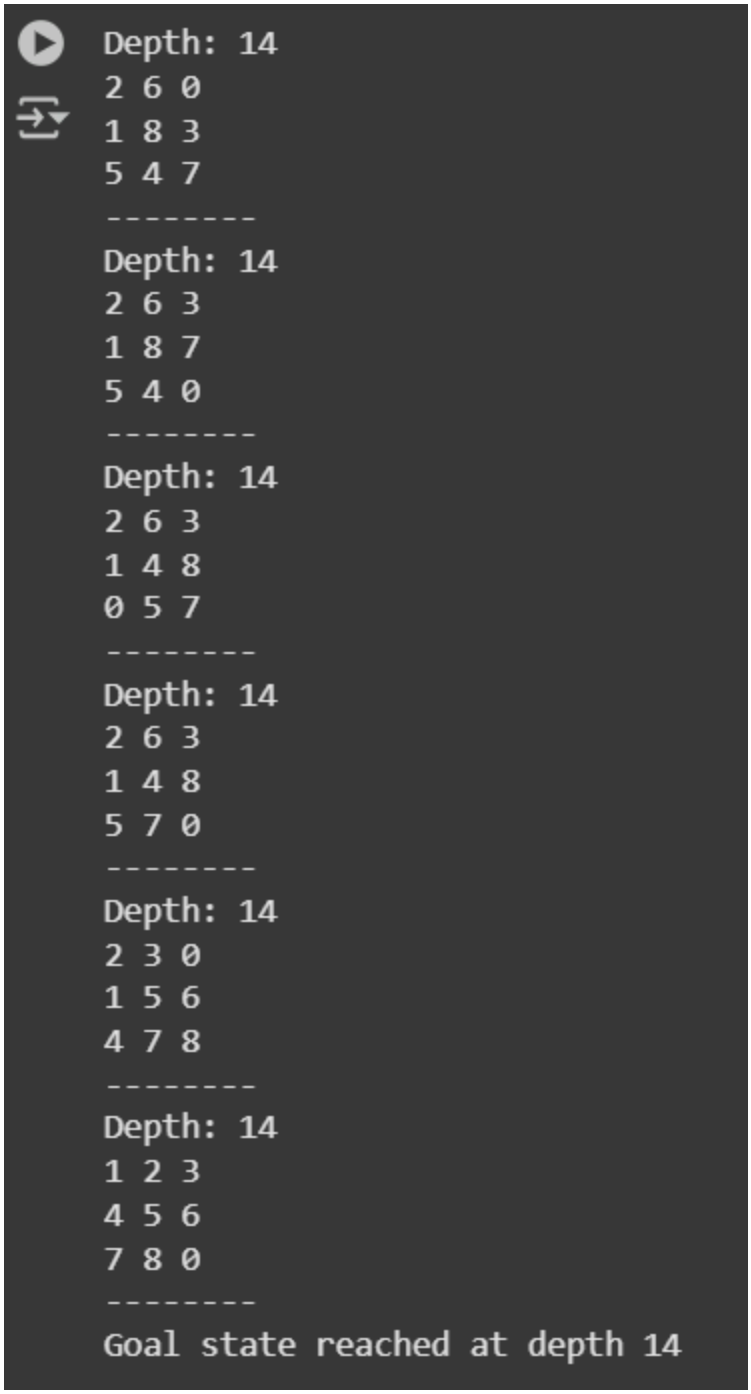
```
    print('No solution found (BFS Brute Force reached depth limit)')

# Driver Code
if __name__ == '__main__':
    start = [[1, 2, 3], [4, 0, 5], [6, 7, 8]]  # Initial state
    x, y = 1, 1

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

    solve_puzzle_bfs(start, x, y)
```

Output Snapshots



```
Depth: 14
2 6 0
1 8 3
5 4 7
--------
Depth: 14
2 6 3
1 8 7
5 4 0
--------
Depth: 14
2 6 3
1 4 8
0 5 7
--------
Depth: 14
2 6 3
1 4 8
5 7 0
--------
Depth: 14
2 3 0
1 5 6
4 7 8
--------
Depth: 14
1 2 3
4 5 6
7 8 0
--------
Goal state reached at depth 14
```