8 Puzzle Solver

Name: Archy Mittal

Roll No: 2024011004000046

Subject: Introduction to AI

To: Bikki Gupta Sir

Introduction:

The code provided solves the classic 8-puzzle problem using the Depth-First Search (DFS) algorithm. The puzzle consists of a 3x3 grid, where one of the tiles is missing (represented by 0), and the goal is to move the tiles around to reach a predefined goal state, usually:

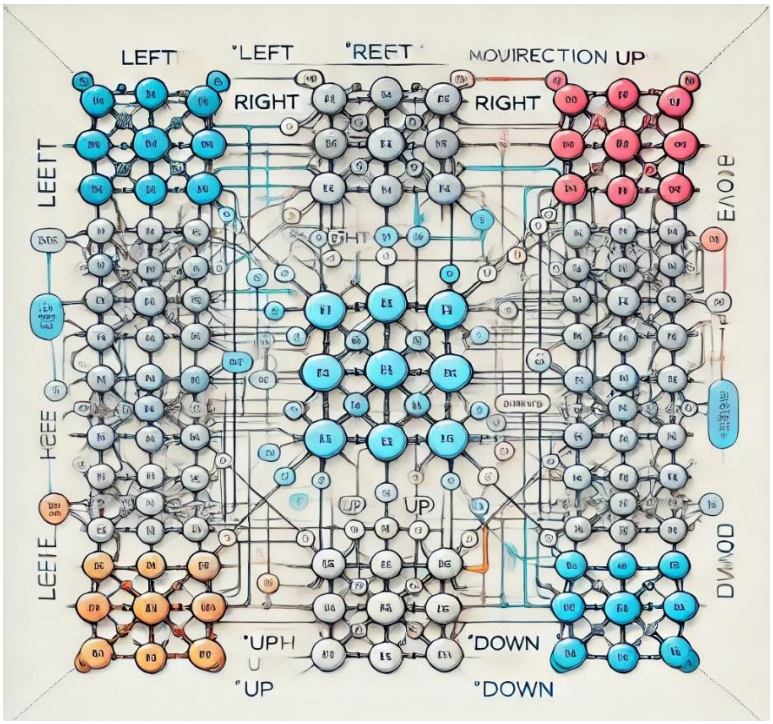
1 2 3
4 5 6
7 8 0

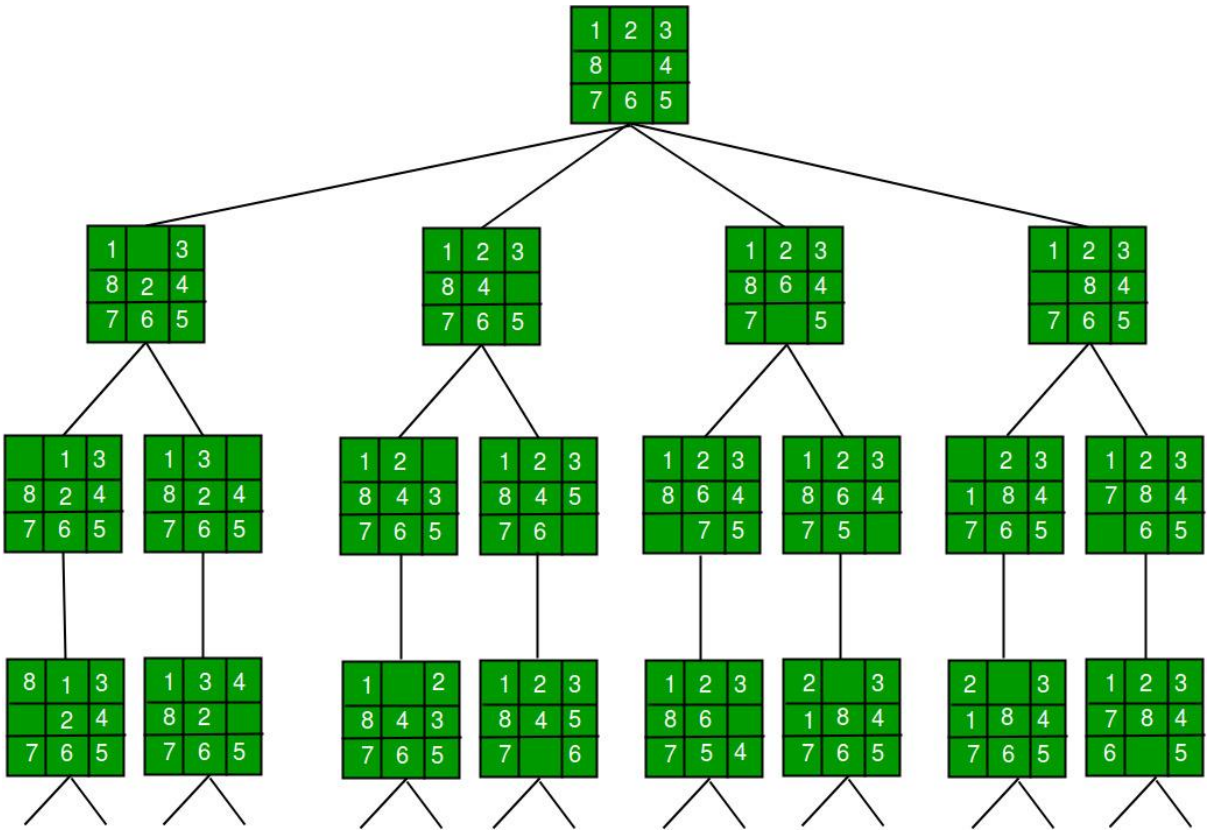
The algorithm explores different states of the puzzle by shifting the tiles in four possible directions (Left, Right, Up, Down). It does so recursively by using a stack to simulate the DFS traversal and explores the states layer by layer. This implementation is also visualized using a graph, where each node represents a puzzle state, and the edges represent the moves made to transition from one state to another.

In this specific version:

1. **Depth-First Search (DFS) is used to find the solution, limiting the search to a depth of 2.**
2. **Graph visualization is used to display the transition from one state to another.**
3. **The program prints each state of the puzzle along with the path it took to reach it and then visualizes the state transitions as a directed graph using NetworkX.**

This visualization helps in understanding how the DFS algorithm works and how the state space of the puzzle evolves as the tiles are moved.


LEFT
RIGHT
UP
DOWN



Key Components:

1. **PuzzleState Class: Stores the state of the puzzle, including the current configuration of the board, the position of the empty tile, the depth (number of moves taken), and the path taken to reach the current state.**
2. **Functions:**
    - **is_goal_state(board): Checks if the given board matches the goal state.**
    - **is_valid(x, y): Ensures that the tile position remains within the bounds of the 3x3 grid.**
    - **print_board(board): Prints the current board configuration.**
    - **solve_puzzle_dfs(start, x, y): The main DFS function, which explores the puzzle states, generates the transitions, and adds them to a directed graph. It also limits the search depth to 2 for efficiency.**
3. **State Transition Graph:**
    - **The DFS explores the puzzle states and records each state transition.**
    - **NetworkX is used to create and visualize the directed graph of the states and transitions.**
4. **DFS Logic:**
    - **The DFS starts with the initial state and explores all possible moves from the current position of the empty space.**
    - **It pushes new states onto the stack, ensuring that previously visited states are not revisited.**
    - **The DFS continues until it finds 3 solutions or explores all possible states within the specified depth limit (depth 2 in this case).**
5. **Graph Visualization:**
    - **The transitions between states are added as edges in the graph.**
    - **The spring layout is used to display the graph visually, with the depth of each state indicated as a label.**

Explanation of DFS Process:

1. **Start from the initial configuration of the puzzle.**
2. **Explore all 4 possible directions for the empty tile (up, down, left, right).**
3. **Swap the empty tile with a neighboring tile to create a new configuration.**
4. **Keep track of visited configurations to avoid revisiting the same state.**

5. **Stop the DFS search after reaching a depth of 2 or after finding 3 solutions.**
6. **The graph visualizes how the algorithm traversed the state space to find solutions.**

How it Works:

- **The program starts by displaying the initial state.**
- **It then performs DFS to explore possible moves and transitions between states.**
- **The transition graph is constructed, and at the end, it is visualized, showing the depth and moves made at each step.**

This approach is ideal for visualizing how DFS works and for solving simple cases of the 8-puzzle. The graph shows the state space and how the puzzle evolves during the search.

Example Output:

- **The initial state of the puzzle will be displayed.**
- **The DFS will explore states up to a depth of 2 and will print out each state configuration.**
- **If a solution is found, the path to the solution will be shown.**
- **The final state transition graph will be displayed using Matplotlib**

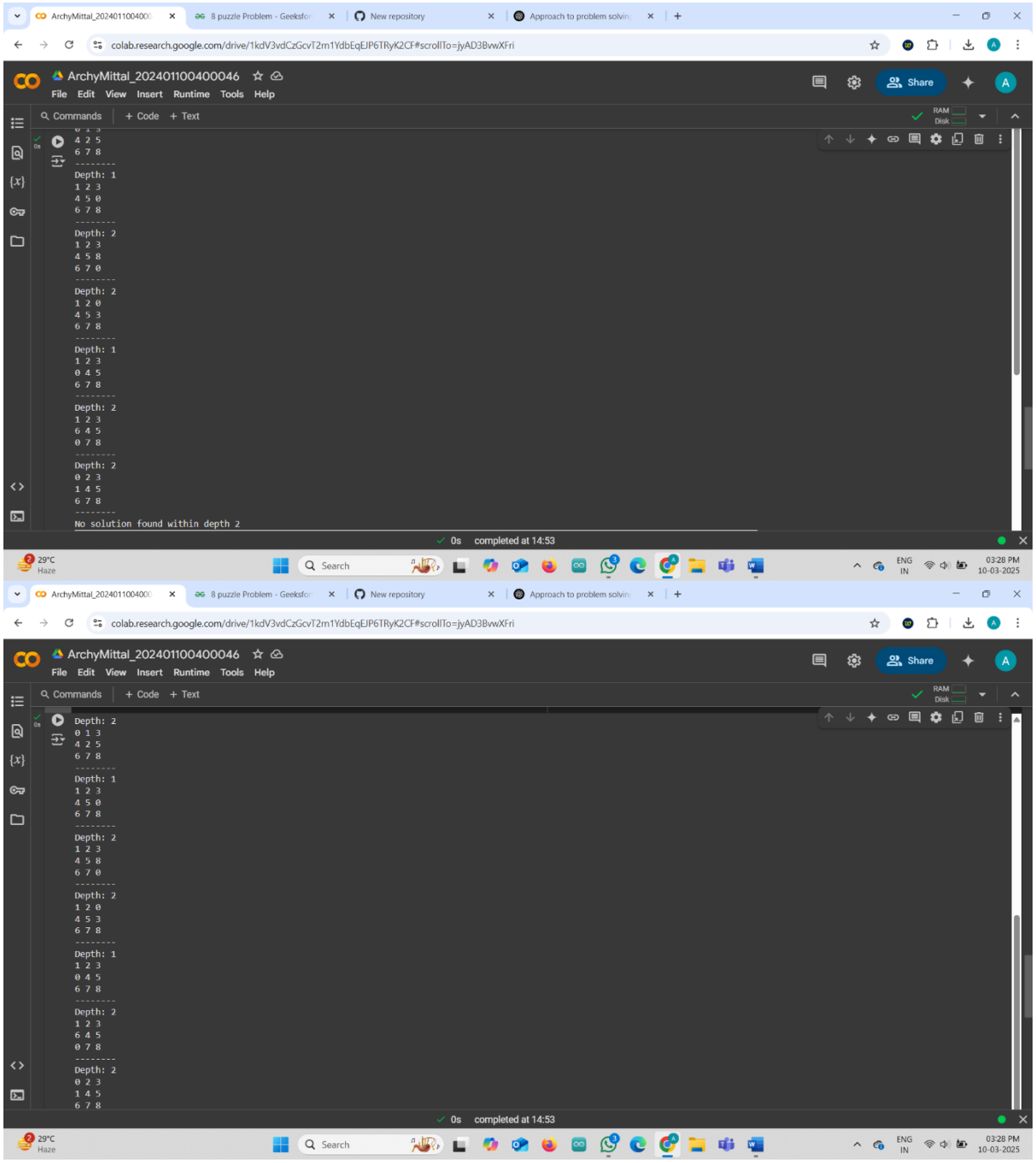
CODE

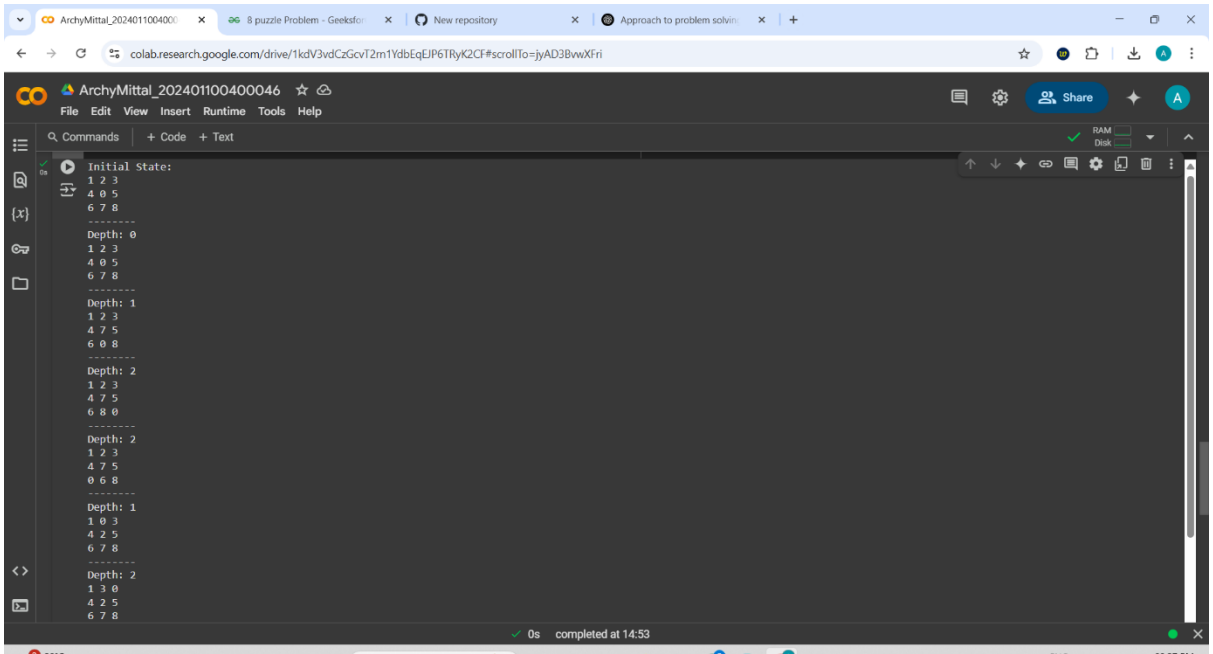
```
from collections import deque
import networkx as nx
import matplotlib.pyplot as plt

# 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=None):
        self.board = board  # Current puzzle board
        self.x = x  # X-coordinate of the empty tile
        self.y = y  # Y-coordinate of the empty tile
        self.depth = depth  # Depth of the current state
        self.path = path if path else []  # Path taken to reach this state

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

# 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("--------")

# Depth-First Search to solve the 8-puzzle problem
def solve_puzzle_dfs(start, x, y):
    stack = []  # Stack for DFS
    visited = set()  # Set to track visited states
    solutions = []  # Store up to 3 solutions
    state_graph = nx.DiGraph()  # Directed graph to store state transitions

    # Add the initial state to the stack and mark it as visited
```

```
    stack.append(PuzzleState(start, x, y, 0, []))
    visited.add(tuple(map(tuple, start)))

    # Loop for DFS
    while stack and len(solutions) < 3:  # Stop after finding 3 solutions
        curr = stack.pop()

        # Limit search depth to 2
        if curr.depth > 2:
            continue

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

        # Add the current board as a node in the graph
        state_tuple = tuple(map(tuple, curr.board))
        state_graph.add_node(state_tuple, label=f"Depth: {curr.depth}")

        # Check if goal state is reached
        if is_goal_state(curr.board):
            solutions.append(curr)
            print(f'Goal state reached at depth {curr.depth}')
            print("Path to solution:", " -> ".join(curr.path))
            continue

        # Explore 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]  # Copy the 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 +
[move_names[i]]))

                    # Add an edge in the graph representing this move
                    state_graph.add_edge(state_tuple, board_tuple, label=move_names[i])

    # Display the final graph
    if not solutions:
        print('No solution found within depth 2')
```

```
    # Draw the graph with node labels
    plt.figure(figsize=(10, 8))
    pos = nx.spring_layout(state_graph)  # Position nodes using spring layout
    labels = nx.get_edge_attributes(state_graph, 'label')
    nx.draw(state_graph, pos, with_labels=True, node_size=2000, node_color="lightblue",
font_size=8, font_weight='bold')
    nx.draw_networkx_edge_labels(state_graph, pos, edge_labels=labels)
    plt.title("8-Puzzle State Transition Graph")
    plt.show()

# Driver Code
if _name_ == '_main_':
    start = [[1, 2, 3], [4, 0, 5], [6, 7, 8]]  # Initial state
    x, y = 1, 1  # Position of the empty tile

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

    solve_puzzle_dfs(start, x, y)
```

OUTPUT FOR 8 PUZZLE GAME




ArchyMittal_202401100400046
File Edit View Insert Runtime Tools Help
Share
Commands + Code + Text
Initial State:
1 2 3
4 0 5
6 7 8

Depth: 0
1 2 3
4 0 5
6 7 8

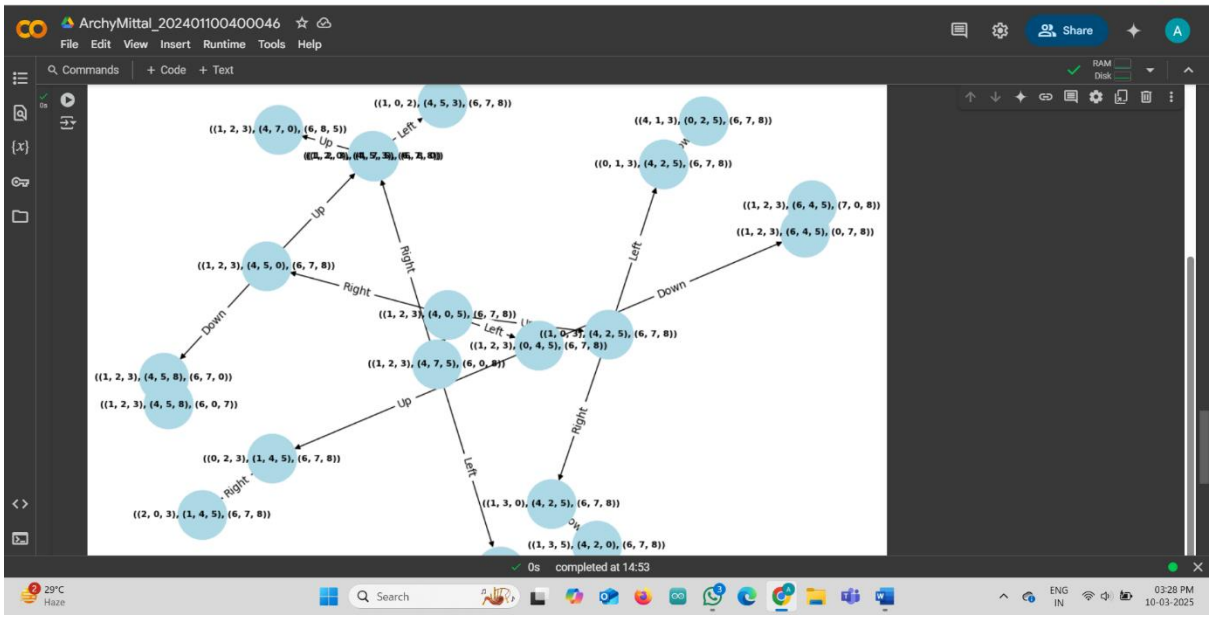
Depth: 1
1 2 3
4 7 5
6 0 8

Depth: 2
1 2 3
4 7 5
6 8 0

Depth: 2
1 2 3
4 7 5
0 6 8

Depth: 1
1 0 3
4 2 5
6 7 8

Depth: 2
1 3 0
4 2 5
6 7 8
0s completed at 14:53


ArchyMittal_202401100400046
File Edit View Insert Runtime Tools Help
Share
Commands + Code + Text
((1, 0, 2), (4, 5, 3), (6, 7, 8))
((1, 2, 3), (4, 7, 0), (6, 8, 5))
((4, 1, 3), (0, 2, 5), (6, 7, 8))
((0, 1, 3), (4, 2, 5), (6, 7, 8))
((1, 2, 3), (6, 4, 5), (7, 0, 8))
((1, 2, 3), (6, 4, 5), (0, 7, 8))
((1, 2, 3), (4, 5, 0), (6, 7, 8))
((1, 2, 3), (4, 0, 5), (6, 7, 8))
((1, 0, 3), (4, 2, 5), (6, 7, 8))
((1, 2, 3), (0, 4, 5), (6, 7, 8))
((1, 2, 3), (4, 7, 5), (6, 0, 8))
((1, 2, 3), (4, 5, 8), (6, 7, 0))
((1, 2, 3), (4, 5, 8), (6, 0, 7))
((0, 2, 3), (1, 4, 5), (6, 7, 8))
((2, 0, 3), (1, 4, 5), (6, 7, 8))
((1, 3, 0), (4, 2, 5), (6, 7, 8))
((1, 3, 5), (4, 2, 0), (6, 7, 8))
Up
Left
Right
Down
0s completed at 14:53

CREDIT

Special Thanks to
Bikki Gupta Sir