

Title:

“8-Puzzle Solver (using Breadth-First Search (BFS))”

Author:

Sudhanshu Singh (202401100400192)

Date:

10-03-2025

Course:

Introduction to AI

Class:

CSE-AIML Sec-C

Institution:

Krishna Institute of Engineering and Technology

Introduction

The 8-Puzzle is a classic sliding puzzle problem that involves a 3x3 grid with 8 numbered tiles and one blank space. The goal of the puzzle is to move the tiles around using the blank space until the puzzle reaches a specific goal configuration. The challenge lies in finding the sequence of moves required to achieve the goal configuration from a given initial state.

The goal configuration of the 8-puzzle is typically defined as:

1 2 3

4 5 6

7 8 0

Where 0 represents the blank space.

Problem Description:

Given an initial configuration of the puzzle, the objective is to find the shortest sequence of moves that will rearrange the tiles to match the goal configuration. The movement is restricted to sliding adjacent tiles into the blank space. The tiles can move up, down, left, or right, provided they remain within the bounds of the 3x3 grid.

The problem is a classical example of a search problem in computer science, where we explore possible states and find the path that leads from the initial state to the goal state.

Significance:

The 8-puzzle is often used as an example in artificial intelligence (AI) and computer science courses to teach search algorithms. It provides insights into problem-solving techniques such as state space search, pathfinding algorithms, and how to implement basic search strategies like Breadth-First Search (BFS) and Depth-First Search (DFS).

Methodology

In this project, we implemented an 8-puzzle solver using the Breadth-First Search (BFS) algorithm. BFS is particularly suitable for this problem because it guarantees finding the shortest path from the initial state to the goal state, making it an optimal solution for the 8-puzzle.

Approach:

The 8-puzzle can be viewed as a state space search problem where:

Each configuration of the puzzle is a state.

A transition from one state to another occurs by sliding a tile into the blank space.

The goal is to reach a configuration where the tiles are arranged in the goal state.

Key Steps in the Approach:

1.State Representation:

The state of the puzzle is represented as a tuple of tuples, where each tuple corresponds to a row in the puzzle grid. For example, the initial state ((1, 2, 3), (4, 5, 6), (0, 7, 8)) represents a 3x3 puzzle with the blank space at the bottom-left corner.

2.Valid Moves:

We define the possible moves of the blank space (represented by 0). The blank space can move:

Up: The tile above the blank space moves into the blank space.

Down: The tile below the blank space moves into the blank space.

Left: The tile to the left of the blank space moves into the blank space.

Right: The tile to the right of the blank space moves into the blank space.

Breadth-First Search (BFS):

1.Initialization: BFS begins by adding the initial state to a queue, and we maintain a set of visited states to avoid revisiting states.

2.State Exploration: We repeatedly dequeue the first state from the queue and explore all valid moves (up, down, left, right). Each new valid state is added to the queue, and we keep track of the sequence of moves that led to that state.

3.Goal Test: If we reach the goal state, BFS stops, and the sequence of states (path) leading to the goal is returned.

Input/Output:

The program prompts the user to enter the initial state of the puzzle row by row.

The program uses BFS to solve the puzzle and prints the sequence of states (the solution path) that leads from the initial state to the goal state.

Code

```
from collections import deque

# Goal state for the 8 puzzle
goal_state = ((1, 2, 3), (4, 5, 6), (7, 8, 0)) # 0 represents the blank space

# Directions in which we can move the blank space (up, down, left, right)
directions = [(-1, 0), (1, 0), (0, -1), (0, 1)] # (dx, dy) for up, down, left, right

def is_goal_state(state):
    """Check if the current state matches the goal state."""
    return state == goal_state

def print_state(state):
    """Print the puzzle state in a human-readable format."""
    for row in state:
        print(row)

def find_blank_position(state):
    """Find the position (row, col) of the blank space (represented by 0)."""
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j

def is_valid_move(x, y):
    """Check if the move is within bounds of the 3x3 grid."""
    return 0 <= x < 3 and 0 <= y < 3

def swap(state, pos1, pos2):
    """Swap two elements in the state (pos1 and pos2 are (row, col) tuples)."""
    state_list = [list(row) for row in state] # Convert tuple to list for mutability

    state_list[pos1[0]][pos1[1]], state_list[pos2[0]][pos2[1]] = state_list[pos2[0]][pos2[1]],
state_list[pos1[0]][pos1[1]]

    return tuple(tuple(row) for row in state_list) # Convert back to tuple

def bfs_solver(initial_state):
    """Solve the 8-puzzle using Breadth-First Search (BFS)."""
```

```

# Initialize the queue with the initial state and an empty path
queue = deque([(initial_state, [])])

visited = set([initial_state]) # Set to track visited states

while queue:

    state, path = queue.popleft()

    # If we've reached the goal state, return the solution path
    if is_goal_state(state):
        return path

    # Find the position of the blank space
    x, y = find_blank_position(state)

    # Try all possible moves (up, down, left, right)
    for dx, dy in directions:
        new_x, new_y = x + dx, y + dy

        # Check if the new position is within bounds
        if is_valid_move(new_x, new_y):
            # Swap the blank space with the adjacent tile
            new_state = swap(state, (x, y), (new_x, new_y))

            # If the new state has not been visited before, add it to the queue
            if new_state not in visited:
                visited.add(new_state)

                queue.append((new_state, path + [new_state])) # Append the new state to the path

return None # If no solution is found

def solve_puzzle(initial_state):
    """Solves the puzzle and prints the solution path."""
    solution = bfs_solver(initial_state)

    if solution is not None:
        print("Solution found:")

```

for step in solution:

```
print_state(step)
```

```
print("----")
```

else:

```
print("No solution found.")
```

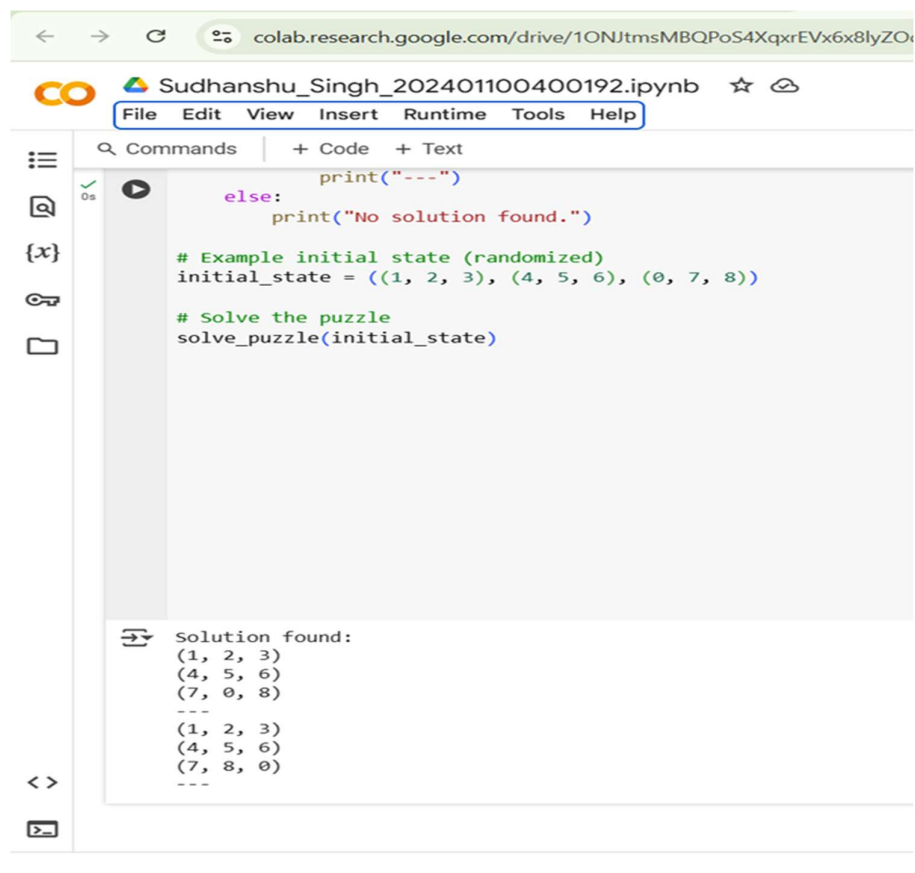
Example initial state (randomized)

```
initial_state = ((1, 2, 3), (4, 5, 6), (0, 7, 8))
```

Solve the puzzle

```
solve_puzzle(initial_state)
```

Output



The screenshot shows a Google Colab notebook interface. The browser address bar at the top displays the URL: `colab.research.google.com/drive/1ONJtmsMBQPoS4XqxrEVx6x8lyZO...`. The notebook title is `Sudhanshu_Singh_202401100400192.ipynb`. The menu bar includes `File`, `Edit`, `View`, `Insert`, `Runtime`, `Tools`, and `Help`. The left sidebar contains icons for file management and execution. The main code area contains the following Python code:

```
print("----")
else:
    print("No solution found.")

# Example initial state (randomized)
initial_state = ((1, 2, 3), (4, 5, 6), (0, 7, 8))

# Solve the puzzle
solve_puzzle(initial_state)
```

The output area at the bottom shows the result of the execution:

```
Solution found:
(1, 2, 3)
(4, 5, 6)
(7, 0, 8)
----
(1, 2, 3)
(4, 5, 6)
(7, 8, 0)
----
```

