B.N.M. Institute of Technology

Autonomous Institution under VTU, Approved by AICTE

Department of Artificial Intelligence and Machine Learning

Vidyayāmruthamashnuthe

Subject: Artificial Intelligence
23AML133

Activity Report on

8-Puzzle Problem using A* algorithm

Submitted in partial fulfillment for the award of degree of

Bachelor of Engineering in

Artificial Intelligence and Machine Learning

Submitted by
Twisha G (1BG23AI115)
Vedant Jain(1BG23AI121)
Vibha Kashyap (1BG23AI122)

Faculty In-charge
Dr. Kakoli Bora
Associate professor
Department of AIML, BNMIT

Problem Statement

The **8 Puzzle Problem** is a sliding puzzle consisting of a 3x3 grid with numbered tiles from 1 to 8 and one empty space (represented as 0). The tiles can slide into the empty space to rearrange the board configuration. The objective is to reach a specified goal state from a given initial configuration.

Initial State Example:

1 2 3
4 0 5
6 7 8

Goal State Example:

1 2 3
4 5 6
7 8 0

Objective:

- Develop a solution using the *A* (A-Star) algorithm* to find the shortest path from the initial state to the goal state.
- Implement the algorithm in Python and validate its efficiency with sample outputs.

Approach Used to Solve the Problem:

To solve the 8 Puzzle Problem, we utilize the *A* search algorithm*, a popular heuristic search method in artificial intelligence. This algorithm is well-suited for finding the shortest path in a search space, combining the cost of reaching the current node (g(n)g(n)g(n)) with an estimate of the cost to reach the goal (h(n)h(n)h(n)).

Key Components of the Approach:

1. State Representation:

- Each state of the puzzle is represented as a list of 9 elements, where the empty space (0) can move up, down, left, or right.

2. Heuristic Function:

- We use the **Manhattan Distance** heuristic, which calculates the sum of the distances of each tile from its goal position.
- The formula is: h(n)=∑(∣xcurrent−xgoal∣+∣ycurrent−ygoal∣)h(n) = \sum (|x_{current} - x_{goal}| + |y_{current} - y_{goal}|)h(n)=∑(∣xcurrent−xgoal

|+|ycurrent−ygoal|) This heuristic is **admissible**, meaning it never overestimates the cost to reach the goal, making A* optimal.

3. Algorithm Description:

- The *A* algorithm* maintains two lists:
    - **Open List**: Stores the nodes to be explored, prioritized by their total cost f(n)f(n)f(n).
    - **Closed List**: Stores the nodes already explored to avoid revisiting them.
- **Steps:**
    1. Initialize the open list with the initial state and an empty closed list.
    2. While the open list is not empty:
        - Extract the node with the lowest total cost f(n)=g(n)+h(n)f(n) = g(n) + h(n)f(n)=g(n)+h(n).
        - If the current node is the goal state, return the solution path.
        - Generate all possible successor states (by moving the empty space).
        - For each successor:
            - Calculate its g(n)g(n)g(n), h(n)h(n)h(n), and f(n)f(n)f(n) values.
            - Add it to the open list if it has not been visited or has a lower cost than a previously found path.

Advantages of A* Algorithm:

- **Optimal and Complete**: Guarantees to find the shortest path if a solution exists.
- **Efficient**: The heuristic function guides the search, reducing the number of explored nodes.

Python Code with Relevant Outputs

```
import tkinter as tk
from tkinter import messagebox
import heapq
import copy

# A* Algorithm for the 8-puzzle problem
class PuzzleNode:
    def _init_(self, state, parent, move, depth, cost):
        self.state = state
        self.parent = parent
        self.move = move
        self.depth = depth
```

```
        self.cost = cost

    def _lt_(self, other):
        return self.cost < other.cost


def is_solvable(puzzle):
    inversions = 0
    flat_puzzle = [tile for row in puzzle for tile in row if tile != 0]
    for i in range(len(flat_puzzle)):
        for j in range(i + 1, len(flat_puzzle)):
            if flat_puzzle[i] > flat_puzzle[j]:
                inversions += 1
    return inversions % 2 == 0

def get_heuristic_cost(state, goal):
    return sum(
        abs(r1 - r2) + abs(c1 - c2)
        for r1, row in enumerate(state)
        for c1, tile in enumerate(row)
        if tile != 0
        for r2, goal_row in enumerate(goal)
        for c2, goal_tile in enumerate(goal_row)
        if tile == goal_tile
    )

def get_possible_moves(state):
    moves = []
    zero_row, zero_col = [(row, col) for row in range(3) for col in range(3) if
state[row][col] == 0][0]

    directions = {"Up": (-1, 0), "Down": (1, 0), "Left": (0, -1), "Right": (0, 1)}
    for move, (dr, dc) in directions.items():
        new_row, new_col = zero_row + dr, zero_col + dc
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_state = copy.deepcopy(state)
            new_state[zero_row][zero_col], new_state[new_row][new_col] =
new_state[new_row][new_col], new_state[zero_row][zero_col]
            moves.append((move, new_state))
    return moves

def solve_puzzle(initial_state, goal_state):
    if not is_solvable(initial_state):
        return None

    open_set = []
    heapq.heappush(open_set, PuzzleNode(initial_state, None, None, 0,
get_heuristic_cost(initial_state, goal_state)))
    visited = set()
```

```
    while open_set:
        current_node = heapq.heappop(open_set)

        if current_node.state == goal_state:
            moves = []
            while current_node.parent is not None:
                moves.append(current_node.move)
                current_node = current_node.parent
            return moves[::-1]

        visited.add(tuple(tuple(row) for row in current_node.state))

        for move, new_state in get_possible_moves(current_node.state):
            if tuple(tuple(row) for row in new_state) not in visited:
                new_cost = current_node.depth + 1 + get_heuristic_cost(new_state,
goal_state)
                heapq.heappush(open_set, PuzzleNode(new_state, current_node, move,
current_node.depth + 1, new_cost))

    return None


# Tkinter GUI
class PuzzleApp:
    def _init_(self, root):
        self.root = root
        self.root.title("8-Puzzle Solver")
        self.root.configure(bg="#F8F8FF")

        # Variables
        self.initial_grid = [[tk.StringVar() for _ in range(3)] for _ in range(3)]
        self.goal_grid = [[tk.StringVar() for _ in range(3)] for _ in range(3)]

        # Title
        tk.Label(root, text="8-Puzzle Solver", font=("Arial", 24, "bold"),
bg="#F8F8FF", fg="#4CAF50").grid(
            row=0, column=0, columnspan=3, pady=(10, 20)
        )

        # Initial State Input
        tk.Label(root, text="Enter Initial State:", font=("Arial", 16, "bold"),
bg="#F8F8FF", fg="#333").grid(
            row=1, column=0, columnspan=3, pady=10
        )
        self.create_grid(self.initial_grid, row_offset=2)

        # Goal State Input
```

```
        tk.Label(root, text="Enter Goal State:", font=("Arial", 16, "bold"),
bg="#F8F8FF", fg="#333").grid(
            row=6, column=0, columnspan=3, pady=10
        )
        self.create_grid(self.goal_grid, row_offset=7)

        # Solve Button
        self.solve_button = tk.Button(
            root,
            text="Solve Puzzle",
            font=("Arial", 14, "bold"),
            bg="#4CAF50",
            fg="white",
            command=self.solve_puzzle,
        )
        self.solve_button.grid(row=11, column=0, columnspan=3, pady=20,
sticky="nsew")

        # Solution Label
        self.solution_label = tk.Label(root, text="", font=("Arial", 14), bg="#F8F8FF",
fg="#333")
        self.solution_label.grid(row=12, column=0, columnspan=3, pady=10)

    def create_grid(self, grid, row_offset):
        """Create a 3x3 grid for either initial or goal state."""
        for r in range(3):
            for c in range(3):
                entry = tk.Entry(
                    self.root,
                    textvariable=grid[r][c],
                    width=5,
                    font=("Arial", 20),
                    justify="center",
                    bg="#FFF8DC",
                    relief="solid",
                    borderwidth=2,
                )
                entry.grid(row=row_offset + r, column=c, padx=5, pady=5)

    def get_grid_state(self, grid):
        """Extract the state from the grid."""
        try:
            return [[int(grid[r][c].get()) for c in range(3)] for r in range(3)]
        except ValueError:
            messagebox.showerror("Input Error", "Please enter valid integers (0-8).")
            return None

    def solve_puzzle(self):
        initial_state = self.get_grid_state(self.initial_grid)
```

```
        goal_state = self.get_grid_state(self.goal_grid)

        if initial_state is None or goal_state is None:
            return

        if not is_solvable(initial_state):
            messagebox.showerror("Unsolvable", "The initial puzzle state is unsolvable.")
            return

        moves = solve_puzzle(initial_state, goal_state)
        if moves:
            self.solution_label.config(text=f"Moves to solve: {', '.join(moves)}",
fg="#4CAF50")
        else:
            self.solution_label.config(text="No solution found.", fg="red")


# Run the application
if _name_ == "_main_":
    root = tk.Tk()
    app = PuzzleApp(root)
     root.mainloop()
```

OUTPUT:

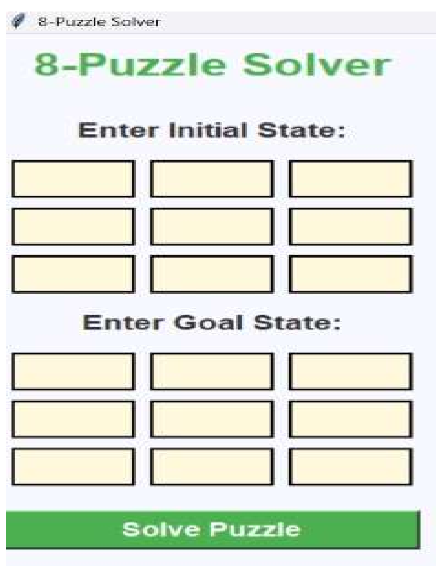

Fig 1: Simple 8-Puzzle Solver GUI

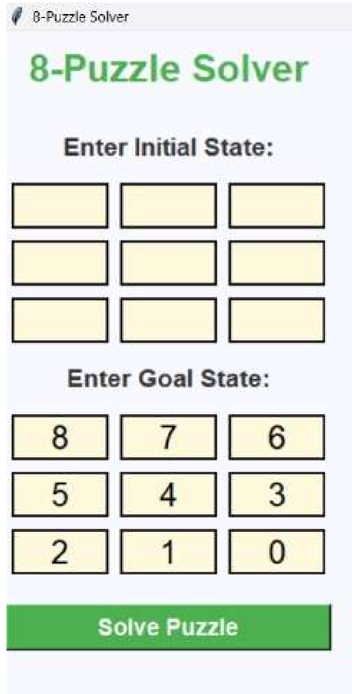

Fig 2: User setting Goal State

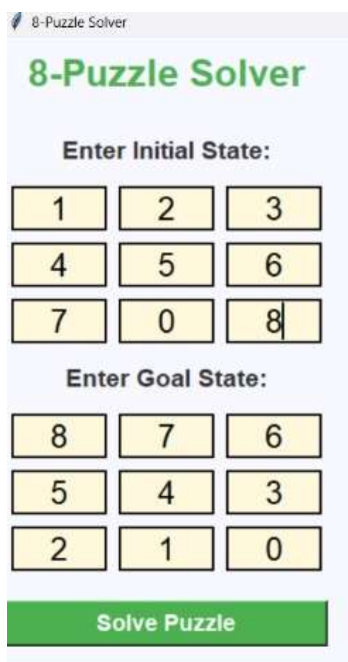

Fig 3: User setting the Initial State

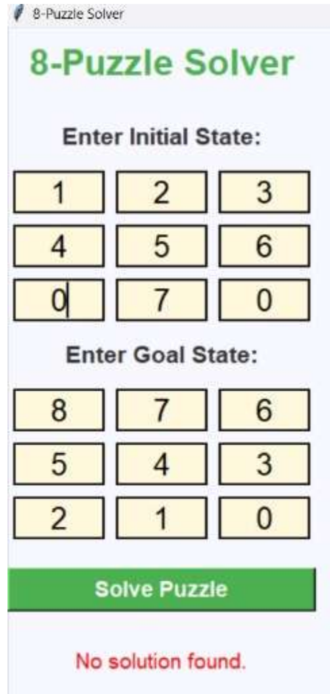

Fig 4: No Solution message pops if no solution can be generated by the given initial state

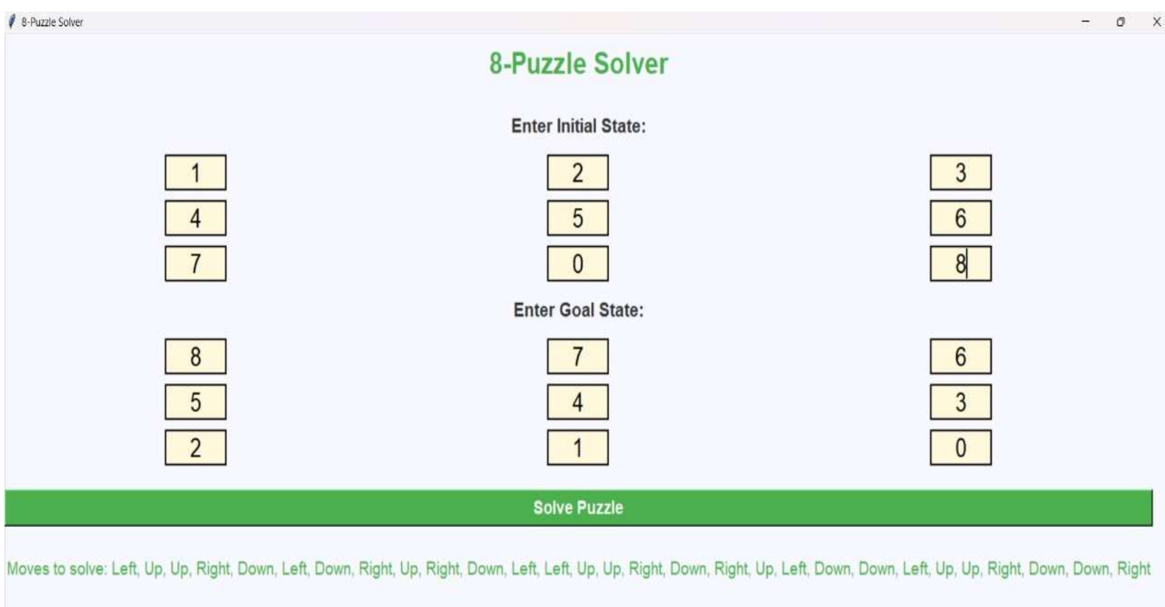

Fig 5: Solution using A* algorithm