8 Puzzle Solver

A PROJECT REPORT
for
Introduction to AI (ID201B)
Session (2024-25)

Submitted by

Harsh Sharma
202410116100084

Harsh Maheshwari
202410116100083

Submitted in partial fulfilment of the
Requirements for the Degree of

MASTER OF COMPUTER APPLICATION

Under the Supervision of
Mr Apoorv Jain

Assistant Professor

Submitted to

DEPARTMENT OF COMPUTER APPLICATIONS
KIET Group of Institutions, Ghaziabad
Uttar Pradesh-201206
(MARCH - 2025)

8 Puzzle Solver

Introduction

The 8-puzzle is a classic problem in artificial intelligence that involves sliding numbered tiles on a 3×3 board to reach a predefined goal state. This project implements an 8-puzzle solver using the A* search algorithm, a widely used heuristic search method. The frontend is developed using React, while the backend utilizes Flask for processing the puzzle-solving logic.

The 8-puzzle problem is a well-known search problem used to test algorithms in artificial intelligence. It consists of a 3×3 grid with eight numbered tiles and a single empty space.

The objective is to move the tiles in a sequence that arranges them in a goal state, typically in ascending order with the empty space in the bottom right corner. This problem can be extended to the 15-puzzle (4×4 grid) or even larger configurations.

This project explores how A* search, a heuristic search algorithm, efficiently finds an optimal solution to the puzzle. The project also examines the impact of heuristics, explores alternative search methods, and

implements an interactive user interface to visualize the solving process.

The importance of the 8-puzzle problem extends beyond recreational challenges. It serves as a fundamental testbed for search algorithms in artificial intelligence. It highlights core AI principles such as problem representation, heuristic design, and optimization techniques. Understanding this problem provides insights into how complex real-world problems, such as pathfinding in robotics and automated planning, can be tackled using AI algorithms.

In addition, this project not only implements the A* search algorithm but also allows users to interact with the puzzle, providing a hands-on learning experience.

By integrating a visual representation with algorithmic problem-solving, this project makes artificial intelligence concepts more accessible and engaging.

Methodology

1. Problem Representation

- The puzzle is represented as a 3×3 grid.
- The empty space (denoted as 0) is used for tile movement.

- Valid moves include swapping the empty tile with its adjacent tiles.
- The state space of the problem consists of all possible board configurations.
- The number of possible configurations is **9! / 2 = 181,440**, considering that half of the states are unsolvable.

2. Algorithm Used: A*

- The A* search algorithm is used to find the optimal path to solve the puzzle.
- The heuristic function used is the **Manhattan Distance**, which calculates the sum of the distances of each tile from its goal position.
- The algorithm explores states with the lowest estimated cost first, leading to efficient solutions.
- A* maintains an **open list** of states to explore and a **closed list** of visited states to avoid redundant calculations.
- The cost function is defined as:
  - f(n) = g(n) + h(n)
  - g(n): Cost to reach the current state.
  - h(n): Heuristic estimate of the remaining cost.

3. Alternative Approaches Considered

Breadth-First Search (BFS)

- Explores all nodes at the current depth before moving to the next level.
- Guaranteed to find the shortest path but is inefficient for large state spaces.

Depth-First Search (DFS)

- Explores as deep as possible before backtracking.
- Not guaranteed to find an optimal solution.

Iterative Deepening A*

- Combines DFS with A* for memory efficiency.
- Useful for large puzzles but slower than standard A*.

4. System Design

Frontend (React)

- Displays the puzzle grid.
- Allows manual tile movement.
- Sends the initial puzzle state to the backend for solving.
- Animates the solution steps sequentially.

- Provides real-time feedback and allows users to test different initial states.

Backend (Flask)

- Receives the puzzle state from the frontend.
- Uses the A* algorithm to compute the solution.
- Returns the sequence of moves to the frontend for visualization.
- Handles invalid input states and ensures solvability before processing.

Code Implementation

Backend Code:

```
from flask import Flask, request, jsonify
from flask_cors import CORS
import heapq
import numpy as np

app = Flask(_name_)
CORS(app)

goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
```

```
def find_zero(board):

    for i in range(3):
        for j in range(3):
            if board[i][j] == 0:
                return i, j

def get_neighbors(x, y):

    moves = [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]
    return [(nx, ny) for nx, ny in moves if 0 <= nx < 3 and 0 <= ny < 3]

def swap_tiles(board, pos1, pos2):

    new_board = [row[:] for row in board]
    x1, y1 = pos1
    x2, y2 = pos2
    new_board[x1][y1], new_board[x2][y2] = new_board[x2][y2], new_board[x1][y1]
```

```
    return new_board

def heuristic(board):

    distance = 0
    for i in range(3):
        for j in range(3):
            if board[i][j] == 0:
                continue
            goal_x, goal_y = divmod(board[i][j] - 1, 3)
            distance += abs(goal_x - i) + abs(goal_y - j)
    return distance

def a_star_solve(board):

    queue = []
    heapq.heappush(queue, (heuristic(board), board, []))
    visited = set()

    while queue:
```

```
        _, state, path = heapq.heappop(queue)

        if state == goal_state:
            return path

        zero_pos = find_zero(state)
        for move in get_neighbors(*zero_pos):
            new_board = swap_tiles(state, zero_pos, move)
            board_tuple = tuple(map(tuple, new_board))

            if board_tuple not in visited:
                visited.add(board_tuple)
                heapq.heappush(queue,
(heuristic(new_board) + len(path), new_board, path +
[new_board]))

    return []


@app.route('/solve', methods=['POST'])
def solve_puzzle():
```

```
    data = request.json
    board = data.get("board")

    if board:
        solution_steps = a_star_solve(board)
        return jsonify({"solution": solution_steps})

    return jsonify({"error": "Invalid board"}), 400

if _name_ == "_main_":
    app.run(debug=True)
```

Frontend Code:

```
import React, { useState, useEffect } from "react";
import { motion } from "framer-motion";
import axios from "axios";
import "./CSS/Home.css";

const initialBoard = () => {
```

```
  let numbers = [...Array(9).keys()];
  numbers.sort(() => Math.random() - 0.5);
  return [
    [numbers[0], numbers[1], numbers[2]],
    [numbers[3], numbers[4], numbers[5]],
    [numbers[6], numbers[7], numbers[8]],
  ];
};

const findZero = (board) => {
  for (let i = 0; i < 3; i++) {
    for (let j = 0; j < 3; j++) {
      if (board[i][j] === 0) return [i, j];
    }
  }
};

const swapTiles = (board, [x1, y1], [x2, y2]) => {
  let newBoard = board.map((row) => [...row]);
```

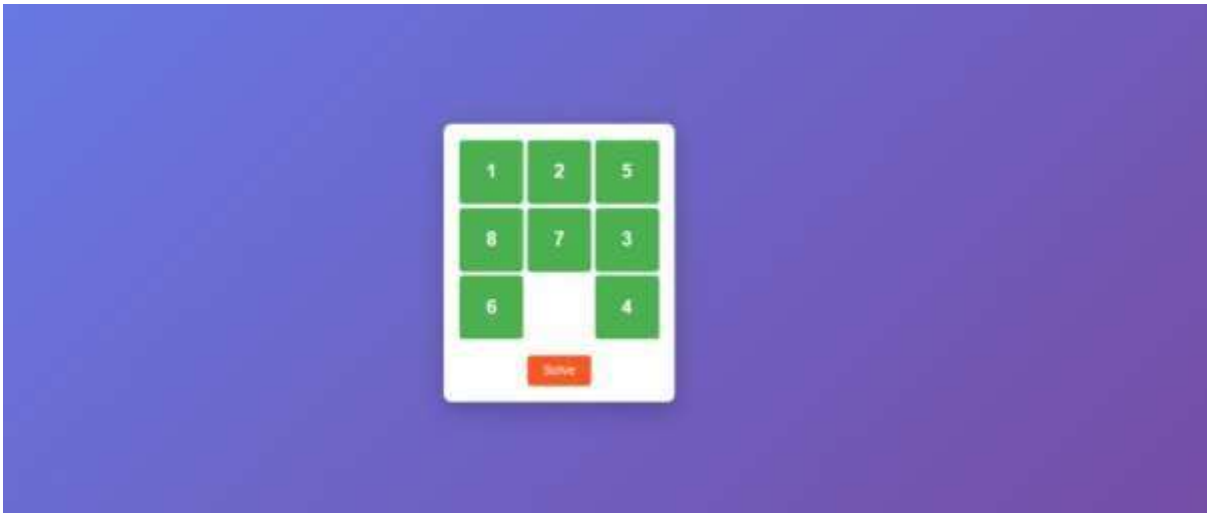
```
    [newBoard[x1][y1], newBoard[x2][y2]] =
[newBoard[x2][y2], newBoard[x1][y1]];
    return newBoard;
  };

  const Home = () => {
    const [board, setBoard] = useState(initialBoard);
    const [solution, setSolution] = useState([]);
    const [step, setStep] = useState(0);

    useEffect(() => {
      setBoard(initialBoard());
    }, []);

    const handleTileClick = (x, y) => {
      const zeroPos = findZero(board);
      const [zx, zy] = zeroPos;
      if (Math.abs(x - zx) + Math.abs(y - zy) === 1) {
        setBoard(swapTiles(board, zeroPos, [x, y]));
      }
```

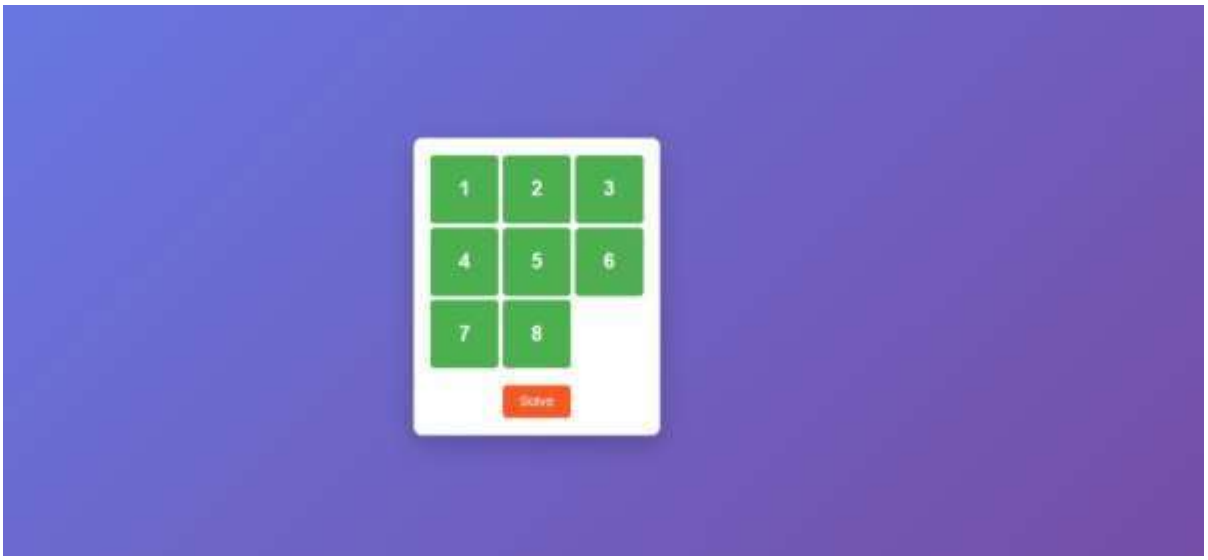
```
  };

  const solvePuzzle = async () => {
    try {
      const response = await
axios.post("http://127.0.0.1:5000/solve", { board });

      setSolution(response.data.solution);
      setStep(0);
    } catch (error)
    {
      console.error("Error fetching solution", error);
    }
  };

  useEffect(() => {
    if (solution.length && step < solution.length) {
      const timer = setTimeout(() => {
        setBoard(solution[step]);
        setStep(step + 1);
```

```
    }, 500);
    return () => clearTimeout(timer);
  }
}, [solution, step]);

return (
  <div className="puzzle-container">
    <div className="grid">
      {board.map((row, i) =>
        row.map((num, j) => (
          <motion.div
            key={num}
            className={tile ${num === 0 ? "empty" : ""}}
            onClick={() => handleTileClick(i, j)}
            whileTap={{ scale: 0.9 }}
          >
            {num !== 0 ? num : ""}
          </motion.div>
        ))
      )}
```

```
        </div>
        <button onClick={solvePuzzle}>Solve</button>
      </div>
    );
  };
export default Home;
```

Output Screenshots:




1
2
3
4
5
6
7
8
Solve