Board Infinity Major Project

AI-Powered 8-Puzzle Solver

Submitted by:

Tuba Mirza

12319516

Lovely Professional University

Table of Contents

1. Introduction

The 8-puzzle is a classic sliding puzzle that consists of a 3x3 grid of numbered tiles from 1 to 8, with one tile missing. The objective is to rearrange the tiles from a random initial configuration into a sorted goal state by sliding tiles into the empty space. This problem serves as an excellent case study in artificial intelligence, particularly for exploring graph traversal and heuristic search algorithms. This report documents the design, implementation, and functionality of an AI-powered 8-Puzzle Solver, which provides both a command-line interface and an interactive web-based game.

2. Project Overview & Objectives

The primary goal of this project is to develop a robust and educational tool for solving the 8-puzzle. The project was developed with two main interfaces in mind:

- A C++ command-line tool for quick, terminal-based solving.
- A feature-rich, interactive web application for a visual and educational user experience.

Key objectives include:
- Implementing two fundamental AI search algorithms: A* (a heuristic-based algorithm) and Backtracking (a form of brute-force search).
- Providing clear, step-by-step visualizations of the solution path.
- Ensuring the application is responsive and can handle complex puzzles without freezing.
- Creating a polished and user-friendly interface for the web version.

3. Core Concepts: Solving the 8-Puzzle

Solving the 8-puzzle involves treating each configuration of the grid as a 'state' in a graph. The goal is to find a path from the initial state to the goal state. The project implements two algorithms to navigate this graph:

A* Search Algorithm

A* is an informed search algorithm that is both complete and optimal. It works by selecting the path that minimizes a cost function, f(n) = g(n) + h(n), where:
- g(n) is the cost of the path from the start node to the current node (n).
- h(n) is a heuristic function that estimates the cost of the cheapest path from n to the goal. For this project, the Manhattan Distance heuristic is used.

Backtracking (Iterative Deepening DFS)

Backtracking is a brute-force algorithm that explores the state space using Depth-First Search (DFS). To prevent getting lost in infinitely deep paths, this project uses Iterative Deepening DFS (IDDFS). It performs a series of depth-limited searches, incrementally increasing the depth limit until a solution is found. While less efficient than A*, it is guaranteed to find the shortest solution path.

4. C++ Command-Line Implementation

The C++ version provides a lightweight, powerful solver that runs directly in the terminal. It is compiled into a single executable and prompts the user to select an algorithm and input the puzzle state. It then prints the solution steps, move count, and execution time.

5. Web-Based Game Implementation

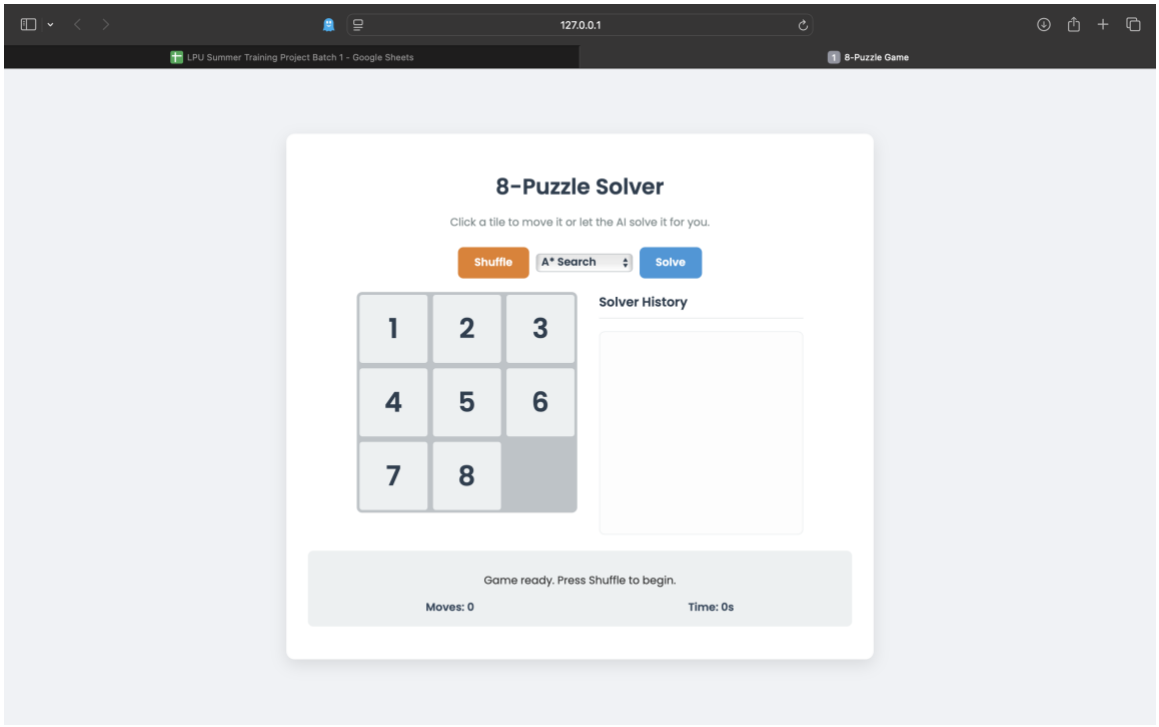
The web application provides a fully interactive and visual experience. Built with HTML, CSS, and JavaScript, it allows users to shuffle the puzzle, attempt to solve it manually, or have the AI solve it with smooth animations. To prevent the browser from freezing during complex solves, the backtracking algorithm is offloaded to a Web Worker, ensuring the UI remains responsive.

6. Live Demo & Screenshots

The live demo can be accessed at: https://mirzasayzz.github.io/AI-Powered-8-Puzzle-Solver/

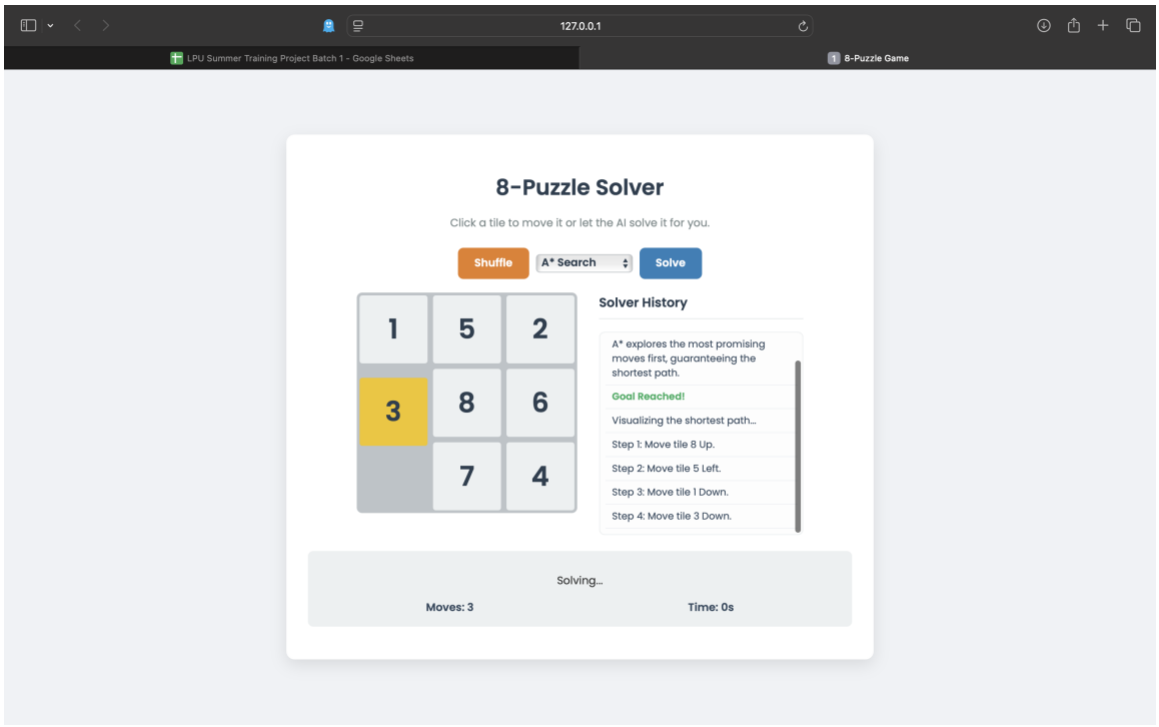
GitHub Repository: https://github.com/mirzasayzz/AI-Powered-8-Puzzle-Solver

Screenshot of the main game interface:



Main game UI

Screenshot of the solver in action (animating the solution):



Solver animation

7. Conclusion & Future Work

This project successfully demonstrates the implementation of classic AI search algorithms to solve the 8-puzzle. Both the C++ and web versions provide effective and educational tools for understanding how these algorithms work. The web application, in particular, offers a polished and engaging user experience.

Future Work

- - Allow users to input custom puzzle configurations in the web UI.
- - Implement additional heuristics for the A* algorithm.
- - Add a timer and move counter for manual solves.

8. Appendix: Source Code

--- C++ (main.cpp) Code ---

```
// Simple 8-puzzle solver demonstrating A* and Backtracking (IDDFS)
// Usage: ./solver <astar|backtracking> 9_numbers (0 represents blank)
// Example: ./solver astar 1 2 3 4 5 6 7 8 0

#include <iostream>
#include <vector>
#include <array>
#include <algorithm>
#include <unordered_set>
#include <unordered_map>
#include <queue>
#include <memory>
#include <chrono>
#include <cmath>
#include <string>

using namespace std;

struct PuzzleState {
    array<int, 9> tiles; // 0 represents blank
    int blankIdx;       // cached index of blank (0)

    PuzzleState() : tiles{}, blankIdx(0) {}
    explicit PuzzleState(const array<int, 9>& t) : tiles(t) {
        blankIdx = find(tiles.begin(), tiles.end(), 0) - tiles.begin();
    }

    bool operator==(const PuzzleState& other) const {
        return tiles == other.tiles;
    }

    bool isGoal() const {
        static const array<int, 9> goal{1,2,3,4,5,6,7,8,0};
        return tiles == goal;
    }

    vector<PuzzleState> neighbors() const {
        static const array<pair<int,int>, 4> moves{ { {0,-1}, {0,1}, {-1,0},
{1,0} } }; // left,right,up,down
        vector<PuzzleState> nbrs;
        int r = blankIdx / 3;
        int c = blankIdx % 3;
        for (const auto& mv : moves) {
            int dr = mv.first;
            int dc = mv.second;
            int nr = r + dr;
            int nc = c + dc;
            if (nr < 0 || nr >= 3 || nc < 0 || nc >= 3) continue;
            int nIdx = nr * 3 + nc;
            PuzzleState next = *this;
```

```
            swap(next.tiles[blankIdx], next.tiles[nIdx]);
            next.blankIdx = nIdx;
            nbrs.push_back(next);
        }
        return nbrs;
    }

    string toString() const {
        string s;
        for (int i = 0; i < 9; ++i) {
            s += to_string(tiles[i]);
            if (i != 8) s += " ";
        }
        return s;
    }
};

struct PuzzleStateHasher {
    size_t operator()(const PuzzleState& s) const noexcept {
        size_t h = 0;
        for (int x : s.tiles) {
            h = h * 31 + x;
        }
        return h;
    }
};

// Utility: check if puzzle is solvable (inversion parity)
bool isSolvable(const PuzzleState& s) {
    int inv = 0;
    for (int i = 0; i < 9; ++i) {
        for (int j = i + 1; j < 9; ++j) {
            if (s.tiles[i] && s.tiles[j] && s.tiles[i] > s.tiles[j]) ++inv;
        }
    }
    return inv % 2 == 0; // For 3x3, solvable if inversions is even
}

// ---------------- Backtracking (DFS) Solver ----------------

bool dfs(const PuzzleState& current, unordered_set<PuzzleState,
PuzzleStateHasher>& visited,
         vector<PuzzleState>& path, int depth, int depthLimit) {
    if (current.isGoal()) return true;
    if (depth >= depthLimit) return false;

    visited.insert(current);
    for (const auto& nxt : current.neighbors()) {
        if (visited.count(nxt)) continue;
        path.push_back(nxt);
        if (dfs(nxt, visited, path, depth + 1, depthLimit)) return true;
        path.pop_back();
    }
    visited.erase(current);
    return false;
```

```
}

vector<PuzzleState> solveBacktracking(const PuzzleState& start, int maxDepth =
50) {
    for (int depthLim = 0; depthLim <= maxDepth; ++depthLim) {
        unordered_set<PuzzleState, PuzzleStateHasher> visited;
        vector<PuzzleState> path{start};
        if (dfs(start, visited, path, 0, depthLim)) return path;
    }
    return {};
}

// ---------------- A* Solver ----------------

int manhattan(const PuzzleState& s) {
    int d = 0;
    for (int idx = 0; idx < 9; ++idx) {
        int val = s.tiles[idx];
        if (val == 0) continue;
        int goalIdx = val - 1;
        d += abs(idx/3 - goalIdx/3) + abs(idx%3 - goalIdx%3);
    }
    return d;
}

struct Node {
    PuzzleState state;
    int g; // cost so far
    int h; // heuristic
    int f; // g + h
    shared_ptr<Node> parent;

    Node(const PuzzleState& s, int g_, int h_, shared_ptr<Node> p)
        : state(s), g(g_), h(h_), f(g_ + h_), parent(std::move(p)) {}
};

struct NodeCmp {
    bool operator()(const shared_ptr<Node>& a, const shared_ptr<Node>& b) const
{
        return a->f > b->f; // min-heap
    }
};

vector<PuzzleState> solveAStar(const PuzzleState& start) {
    priority_queue<shared_ptr<Node>, vector<shared_ptr<Node>>, NodeCmp> open;
    unordered_map<PuzzleState, int, PuzzleStateHasher> bestG;

    auto h0 = manhattan(start);
    auto startNode = make_shared<Node>(start, 0, h0, nullptr);
    open.push(startNode);
    bestG[start] = 0;

    while (!open.empty()) {
        auto node = open.top();
        open.pop();
```

```
        if (node->state.isGoal()) {
            // reconstruct
            vector<PuzzleState> path;
            for (auto n = node; n != nullptr; n = n->parent) path.push_back(n-
>state);
            reverse(path.begin(), path.end());
            return path;
        }

        for (const auto& nbr : node->state.neighbors()) {
            int tentativeG = node->g + 1;
            if (!bestG.count(nbr) || tentativeG < bestG[nbr]) {
                bestG[nbr] = tentativeG;
                auto nbrNode = make_shared<Node>(nbr, tentativeG,
manhattan(nbr), node);
                open.push(nbrNode);
            }
        }
    }
    return {};
}

// ---------------- Terminal I/O ----------------

int main() {
    string alg;
    array<int, 9> tiles{};

    int method = -1;
    cout << "Choose solver: 0 = A* , 1 = Backtracking : ";
    cin >> method;
    alg = (method == 0) ? "astar" : "backtracking";
    cout << "Enter the 9 puzzle numbers separated by spaces (use 0 for blank):
";
    for (int i = 0; i < 9; ++i) {
        cin >> tiles[i];
    }
    PuzzleState start(tiles);

    if (!isSolvable(start)) {
        cout << "The given puzzle is unsolvable.\n";
        return 0;
    }

    vector<PuzzleState> solution;
    auto begin = chrono::steady_clock::now();
    if (alg == "astar") {
        solution = solveAStar(start);
    } else if (alg == "backtracking") {
        solution = solveBacktracking(start, 50);
    } else {
        cout << "Unknown algorithm. Use 'astar' or 'backtracking'.\n";
        return 1;
    }
```

```
    auto end = chrono::steady_clock::now();

    if (solution.empty()) {
        cout << "No solution found within limits.\n";
        return 0;
    }

    cout << "Solution found in " << solution.size() - 1 << " moves.\n";
    cout << "Time taken: " << chrono::duration_cast<chrono::milliseconds>(end-
begin).count() << " ms\n";
    cout << "Steps:\n";
    for (size_t step = 0; step < solution.size(); ++step) {
        cout << "Step " << step << ": " << solution[step].toString() << "\n";
    }

    return 0;
}
```

--- HTML (index.html) Code ---

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>8-Puzzle Game</title>
    <link rel="stylesheet" href="style.css">
    <link
href="https://fonts.googleapis.com/css2?family=Poppins:wght@400;600&display=swa
p" rel="stylesheet">
</head>
<body>
    <div class="container">
        <header>
            <h1>8-Puzzle Solver</h1>
            <p>Click a tile to move it or let the AI solve it for you.</p>
        </header>

        <div class="game-controls">
            <div class="button-group">
                <button id="shuffle-btn">Shuffle</button>
                <select id="algorithm-select">
                    <option value="astar">A* Search</option>
                    <option value="backtracking">Backtracking</option>
                </select>
                <button id="solve-btn">Solve</button>
            </div>
        </div>

        <div class="game-area">
            <div id="puzzle-container">
                <div id="puzzle-grid"></div>
            </div>
            <div id="history-container">
                <h3>Solver History</h3>
                <div id="history-log"></div>
            </div>
        </div>

        <div id="status-panel">
            <p id="status-message">Game ready. Press Shuffle to begin.</p>
            <div class="stats">
                <span id="move-counter">Moves: 0</span>
                <span id="timer">Time: 0s</span>
            </div>
        </div>
    </div>

    <script src="script.js"></script>
</body>
</html>
```

--- CSS (style.css) Code ---

```
body {
    font-family: 'Poppins', sans-serif;
    background-color: #f0f2f5;
    color: #333;
    display: flex;
    justify-content: center;
    align-items: center;
    min-height: 100vh;
    margin: 0;
}

.container {
    background: #ffffff;
    border-radius: 12px;
    box-shadow: 0 8px 24px rgba(0,0,0,0.1);
    padding: 2rem;
    width: 90%;
    max-width: 800px;
    text-align: center;
}

header h1 {
    color: #2c3e50;
    margin-bottom: 0.5rem;
}

header p {
    color: #7f8c8d;
    margin-bottom: 1.5rem;
}

.game-controls {
    display: flex;
    flex-direction: column;
    align-items: center;
    gap: 20px;
    margin-bottom: 20px;
}

.button-group {
    display: flex;
    gap: 10px;
    align-items: center;
}

select#algorithm-select {
    padding: 0.7rem 1rem;
    border-radius: 8px;
    border: 1px solid #bdc3c7;
    font-family: 'Poppins', sans-serif;
    font-size: 1rem;
    font-weight: 600;
```

```
    cursor: pointer;
    background-color: white;
    color: #2c3e50;
}

button {
    padding: 0.7rem 1.5rem;
    border-radius: 8px;
    border: none;
    font-family: 'Poppins', sans-serif;
    font-size: 1rem;
    font-weight: 600;
    cursor: pointer;
    transition: all 0.2s ease;
    color: white;
}

#shuffle-btn {
    background-color: #e67e22;
}
#shuffle-btn:hover {
    background-color: #d35400;
}

#solve-btn {
    background-color: #3498db;
}
#solve-btn:hover {
    background-color: #2980b9;
}

.game-area {
    display: flex;
    justify-content: center;
    gap: 2rem;
    margin-bottom: 1.5rem;
}

#puzzle-container {
    width: 324px;
    height: 324px;
}

#puzzle-grid {
    position: relative;
    width: 100%;
    height: 100%;
    background-color: #bdc3c7;
    border-radius: 8px;
}

.puzzle-tile {
    position: absolute;
    width: 100px;
    height: 100px;
```

```
    margin: 4px;
    background-color: #ecf0f1;
    color: #2c3e50;
    font-size: 2.5rem;
    font-weight: 600;
    display: flex;
    justify-content: center;
    align-items: center;
    border-radius: 4px;
    user-select: none;
    cursor: pointer;
    transition: transform 0.25s ease-in-out;
}

.puzzle-tile.highlight {
    background-color: #f1c40f;
}

#history-container {
    width: 300px;
    text-align: left;
}

#history-container h3 {
    margin-top: 0;
    color: #2c3e50;
    border-bottom: 2px solid #ecf0f1;
    padding-bottom: 0.5rem;
}

#history-log {
    height: 280px;
    overflow-y: auto;
    background: #fdfdfd;
    border: 1px solid #ecf0f1;
    border-radius: 8px;
    padding: 0.5rem;
    font-size: 0.9rem;
    color: #34495e;
}

.log-entry {
    padding: 0.4rem 0.6rem;
    border-bottom: 1px solid #f0f2f5;
}

.log-entry.node-check {
    color: #7f8c8d;
}

.log-entry.goal-found {
    color: #27ae60;
    font-weight: 600;
}
```

```css
#status-panel {
    background-color: #ecf0f1;
    padding: 1rem;
    border-radius: 8px;
    margin-bottom: 1rem;
}

.stats {
    display: flex;
    justify-content: space-around;
    font-weight: 600;
    color: #34495e;
}

.custom-puzzle-controls {
    display: flex;
    justify-content: center;
    gap: 0.5rem;
}

#puzzle-input {
    flex-grow: 1;
    max-width: 400px;
    padding: 0.6rem;
    border: 1px solid #bdc3c7;
    border-radius: 8px;
    font-size: 1rem;
}

#load-puzzle-btn {
    background-color: #27ae60;
    border-color: #27ae60;
    color: white;
}

#load-puzzle-btn:hover {
    background-color: #229954;
}
```

--- JavaScript (script.js) Code ---

```
document.addEventListener('DOMContentLoaded', () => {
    // --- DOM Elements ---
    const grid = document.getElementById('puzzle-grid');
    const shuffleBtn = document.getElementById('shuffle-btn');
    const solveBtn = document.getElementById('solve-btn');
    const algorithmSelect = document.getElementById('algorithm-select');
    const moveCounter = document.getElementById('move-counter');
    const timerDisplay = document.getElementById('timer');
    const statusMessage = document.getElementById('status-message');
    const historyLog = document.getElementById('history-log');

    // --- Game State ---
    let currentState = [];
    let tileElements = {};
    let moveCount = 0;
    let timer = 0;
    let timerInterval = null;
    let isSolving = false;
    const goalState = [1, 2, 3, 4, 5, 6, 7, 8, 0];

    // --- Priority Queue for A* ---
    class PriorityQueue {
        constructor() { this.elements = []; }
        enqueue(element, priority) {
            this.elements.push({ element, priority });
            this.elements.sort((a, b) => a.priority - b.priority);
        }
        dequeue() { return this.elements.shift().element; }
        isEmpty() { return this.elements.length === 0; }
    }

    // --- Game Initialization ---
    function initializeGame() {
        grid.innerHTML = '';
        tileElements = {};
        for (let i = 1; i <= 8; i++) {
            const tileEl = document.createElement('div');
            tileEl.classList.add('puzzle-tile');
            tileEl.textContent = i;
            tileEl.dataset.value = i;
            tileEl.addEventListener('click', () => onTileClick(i));
            grid.appendChild(tileEl);
            tileElements[i] = tileEl;
        }
        loadState(goalState);
        updateStatus('Game ready. Press Shuffle to begin.');
    }

    function loadState(state) {
        currentState = [...state];
        moveCount = 0;
        stopTimer();
```

```
        resetTimer();
        clearHistory();
        updateTilePositions();
        updateMoveCounter();
    }

    // --- Rendering and Animation ---
    function updateTilePositions() {
        for (let i = 0; i < 9; i++) {
            const tileValue = currentState[i];
            if (tileValue === 0) continue;
            const tileEl = tileElements[tileValue];
            const x = i % 3;
            const y = Math.floor(i / 3);
            tileEl.style.transform = `translate(${x * 108}px, ${y * 108}px)`;
        }
    }

    // --- User Interaction ---
    function onTileClick(tileValue) {
        if (isSolving) return;
        const tileIndex = currentState.indexOf(tileValue);
        const blankIndex = currentState.indexOf(0);
        const [row, col] = [Math.floor(tileIndex / 3), tileIndex % 3];
        const [blankRow, blankCol] = [Math.floor(blankIndex / 3), blankIndex %
3];

        if (Math.abs(row - blankRow) + Math.abs(col - blankCol) === 1) {
            if (!timerInterval) startTimer();
            moveCount++;
            swapAndAnimate(tileIndex, blankIndex);
            if (isSolved()) {
                stopTimer();
                updateStatus(`Congratulations! Solved in ${moveCount} moves.`);
            }
        }
    }

    async function swapAndAnimate(indexA, indexB, isSolverMove = false,
movedTile) {
        [currentState[indexA], currentState[indexB]] = [currentState[indexB],
currentState[indexA]];
        updateTilePositions();
        if (isSolverMove) {
            const tileEl = tileElements[movedTile];
            tileEl.classList.add('highlight');
            await new Promise(resolve => setTimeout(resolve, 250));
            tileEl.classList.remove('highlight');
        }
        updateMoveCounter();
    }

    // --- Game Logic & Controls ---
    function isSolved() {
        return JSON.stringify(currentState) === JSON.stringify(goalState);
```

```
    }

    function shuffle() {
        let puzzle = [...goalState];
        for (let i = 0; i < 150; i++) {
            const neighbors = getNeighbors(puzzle).states;
            puzzle = neighbors[Math.floor(Math.random() * neighbors.length)];
        }
        loadState(puzzle);
        updateStatus('Shuffled! Your turn to solve.');
    }

    // --- AI Solvers ---
    async function solve() {
        if (isSolving) return;
        if (isSolved()) {
            updateStatus('Puzzle is already solved!');
            return;
        }
        isSolving = true;
        setControls(false);
        updateStatus('Solving...');
        clearHistory();

        const algorithm = algorithmSelect.value;

        if (algorithm === 'backtracking') {
            // Use web worker for heavy backtracking search
            const worker = new Worker('solver_worker.js');
            const startTime = performance.now();
            worker.postMessage({ state: currentState, algorithm: 'backtracking'
});

            worker.onmessage = async (e) => {
                const { status, path, message } = e.data;
                if (status === 'done' && path) {
                    const duration = ((performance.now() - startTime) /
1000).toFixed(2);
                    await visualizeSolution(path);
                    updateStatus(`Solved with BACKTRACKING in ${duration}s -
${path.length - 1} moves.`);
                } else if (status === 'error') {
                    updateStatus(`Error: ${message}`);
                }
                worker.terminate();
                isSolving = false;
                setControls(true);
            };
            return;
        }

        // Default to in-thread A* (fast)
        const path = solveAStar(currentState);
        if (path && path.length > 0) {
            await visualizeSolution(path);
```

```
            updateStatus(`Solved with A* in ${path.length - 1} moves.`);
        } else {
            updateStatus('No solution found. The puzzle might be too complex or
unsolvable.');
        }
        isSolving = false;
        setControls(true);
    }

    function getNeighbors(state) {
        const moves = [];
        const blankIndex = state.indexOf(0);
        const [row, col] = [Math.floor(blankIndex / 3), blankIndex % 3];
        const directions = { 'Up': [-1, 0], 'Down': [1, 0], 'Left': [0, -1],
'Right': [0, 1] };

        for (const [name, [dr, dc]] of Object.entries(directions)) {
            const newRow = row + dr, newCol = col + dc;
            if (newRow >= 0 && newRow < 3 && newCol >= 0 && newCol < 3) {
                const newIndex = newRow * 3 + newCol;
                const newState = [...state];
                const movedTile = newState[newIndex];
                [newState[blankIndex], newState[newIndex]] =
[newState[newIndex], newState[blankIndex]];
                moves.push({ state: newState, movedTile, direction: name });
            }
        }
        return { states: moves.map(m => m.state), moves: moves };
    }

    function manhattan(state) {
        let distance = 0;
        for (let i = 0; i < 9; i++) {
            if (state[i] !== 0) {
                const goalIndex = state[i] - 1;
                const [row, col] = [Math.floor(i / 3), i % 3];
                const [goalRow, goalCol] = [Math.floor(goalIndex / 3),
goalIndex % 3];
                distance += Math.abs(row - goalRow) + Math.abs(col - goalCol);
            }
        }
        return distance;
    }

    // --- A* Solver ---
    function solveAStar(initialState) {
        const frontier = new PriorityQueue();
        const initialNode = { state: initialState, parent: null, movedTile:
null, direction: null };
        frontier.enqueue(initialNode, manhattan(initialState));

        const cameFrom = { [JSON.stringify(initialState)]: null };
        const costSoFar = { [JSON.stringify(initialState)]: 0 };

        addHistoryLog('Starting A* Search...', 'title');
```

```
        addHistoryLog('A* explores the most promising moves first, guaranteeing
the shortest path.', 'info');

        while (!frontier.isEmpty()) {
            const current = frontier.dequeue();
            const currentStateStr = JSON.stringify(current.state);

            if (currentStateStr === JSON.stringify(goalState)) {
                addHistoryLog('Goal Reached!', 'goal-found');
                let path = [];
                let temp = current;
                while (temp) {
                    path.unshift(temp);
                    temp = temp.parent;
                }
                return path;
            }

            const { moves } = getNeighbors(current.state);
            for (const move of moves) {
                const newCost = costSoFar[currentStateStr] + 1;
                const nextStateStr = JSON.stringify(move.state);

                if (cameFrom[nextStateStr] === undefined || newCost <
costSoFar[nextStateStr]) {
                    costSoFar[nextStateStr] = newCost;
                    const priority = newCost + manhattan(move.state);
                    const newNode = { state: move.state, parent: current,
movedTile: move.movedTile, direction: move.direction };
                    frontier.enqueue(newNode, priority);
                    cameFrom[nextStateStr] = current;
                }
            }
        }
        return null; // No solution found
    }

    // --- Backtracking Solver (DFS) ---
    function solveBacktracking(initialState) {
        let solutionPath = [];
        const visited = new Set();
        const maxDepth = 35; // Safety limit to prevent infinite loops in very
hard puzzles

        addHistoryLog('Starting Backtracking Search...', 'title');
        addHistoryLog('Backtracking explores one path deeply, then backtracks
if it hits a dead end.', 'info');

        function dfs(path) {
            if (solutionPath.length > 0) return;

            const current = path[path.length - 1];
            const currentStateStr = JSON.stringify(current.state);

            if (currentStateStr === JSON.stringify(goalState)) {
```

```
                addHistoryLog('Goal Reached!', 'goal-found');
                solutionPath = [...path]; // Create a copy of the found path
                return;
            }

            if (path.length > maxDepth) {
                return;
            }

            const { moves } = getNeighbors(current.state);
            for (const move of moves) {
                const nextStateStr = JSON.stringify(move.state);
                if (!visited.has(nextStateStr)) {
                    visited.add(nextStateStr); // Mark as visited before
recursing
                    const newNode = { state: move.state, parent: current,
movedTile: move.movedTile, direction: move.direction };
                    path.push(newNode);

                    addHistoryLog(`(Depth ${path.length - 1}) Trying: Move
${move.movedTile} ${move.direction}`, 'node-check');
                    dfs(path);

                    if (solutionPath.length > 0) return;

                    path.pop(); // Backtrack
                }
            }
        }

        const initialPath = [{ state: initialState, parent: null, movedTile:
null, direction: null }];
        visited.add(JSON.stringify(initialState));
        dfs(initialPath);

        if (solutionPath.length === 0) {
            addHistoryLog('No solution found within the depth limit.', 'info');
        }

        return solutionPath;
    }

    async function visualizeSolution(path) {
        stopTimer();
        addHistoryLog('Visualizing the shortest path...', 'title');
        for (let i = 0; i < path.length - 1; i++) {
            const step = path[i + 1];
            const oldBlank = path[i].state.indexOf(0);
            moveCount = i + 1;
            addHistoryLog(`Step ${moveCount}: Move tile ${step.movedTile}
${step.direction}.`, 'move');
            await swapAndAnimate(oldBlank, step.state.indexOf(0), true,
step.movedTile);
        }
    }
```

```
    // --- UI Updates & Helpers ---
    function setControls(enabled) {
        shuffleBtn.disabled = !enabled;
        solveBtn.disabled = !enabled;
        algorithmSelect.disabled = !enabled;
    }

    function updateStatus(msg) { statusMessage.textContent = msg; }
    function updateMoveCounter() { moveCounter.textContent = `Moves:
${moveCount}`; }

    function startTimer() {
        if (timerInterval) return;
        timer = 0;
        timerInterval = setInterval(() => {
            timer++;
            timerDisplay.textContent = `Time: ${timer}s`;
        }, 1000);
    }

    function stopTimer() {
        clearInterval(timerInterval);
        timerInterval = null;
    }

    function resetTimer() {
        timer = 0;
        timerDisplay.textContent = `Time: ${timer}s`;
    }

    function addHistoryLog(message, className = '') {
        const entry = document.createElement('div');
        entry.classList.add('log-entry');
        if (className) entry.classList.add(className);
        entry.textContent = message;
        historyLog.appendChild(entry);
        historyLog.scrollTop = historyLog.scrollHeight;
    }

    function clearHistory() { historyLog.innerHTML = ''; }

    // --- Event Listeners ---
    shuffleBtn.addEventListener('click', shuffle);
    solveBtn.addEventListener('click', solve);

    // --- Initial Load ---
    initializeGame();
});
```

--- JavaScript Web Worker (solver_worker.js) Code ---

```
// Web Worker to run heavy puzzle solving algorithms off the main UI thread

/*
  Expected inbound message format:
  {
      state: Array<number>,  // length 9, 0 represents blank
      algorithm: 'backtracking' | 'astar'
  }

  Outbound message format:
  {
      status: 'done' | 'progress' | 'error',
      path?: Array<{state: number[], movedTile: number, direction: string}>,
      message?: string,
      nodesSearched?: number
  }
*/

self.addEventListener('message', (e) => {
    const { state, algorithm } = e.data;
    try {
        if (algorithm === 'backtracking') {
            const path = solveBacktracking(state);
            self.postMessage({ status: 'done', path });
        } else if (algorithm === 'astar') {
            const path = solveAStar(state);
            self.postMessage({ status: 'done', path });
        } else {
            self.postMessage({ status: 'error', message: 'Unknown algorithm'
});
        }
    } catch (err) {
        self.postMessage({ status: 'error', message: err.message || String(err)
});
    }
});

function getNeighbors(state) {
    const moves = [];
    const blankIndex = state.indexOf(0);
    const [row, col] = [Math.floor(blankIndex / 3), blankIndex % 3];
    const directions = { 'Up': [-1, 0], 'Down': [1, 0], 'Left': [0, -1],
'Right': [0, 1] };

    for (const [name, [dr, dc]] of Object.entries(directions)) {
        const newRow = row + dr, newCol = col + dc;
        if (newRow >= 0 && newRow < 3 && newCol >= 0 && newCol < 3) {
            const newIndex = newRow * 3 + newCol;
            const newState = [...state];
            const movedTile = newState[newIndex];
            [newState[blankIndex], newState[newIndex]] = [newState[newIndex],
newState[blankIndex]];
```

```
            moves.push({ state: newState, movedTile, direction: name });
        }
    }
    return moves;
}

function manhattan(state) {
    let distance = 0;
    for (let i = 0; i < 9; i++) {
        if (state[i] !== 0) {
            const goalIndex = state[i] - 1;
            const [row, col] = [Math.floor(i / 3), i % 3];
            const [goalRow, goalCol] = [Math.floor(goalIndex / 3), goalIndex %
3];
            distance += Math.abs(row - goalRow) + Math.abs(col - goalCol);
        }
    }
    return distance;
}

// Simple priority queue for A* inside worker
class PQ {
    constructor() { this.arr = []; }
    push(node, priority) {
        this.arr.push({ node, priority });
        this.arr.sort((a, b) => a.priority - b.priority);
    }
    pop() { return this.arr.shift().node; }
    isEmpty() { return this.arr.length === 0; }
}

function solveAStar(initialState) {
    const frontier = new PQ();
    const initialNode = { state: initialState, parent: null, movedTile: null,
direction: null };
    frontier.push(initialNode, manhattan(initialState));

    const cameFrom = new Map();
    const costSoFar = new Map();
    const key = JSON.stringify(initialState);
    cameFrom.set(key, null);
    costSoFar.set(key, 0);

    while (!frontier.isEmpty()) {
        const current = frontier.pop();
        const currentKey = JSON.stringify(current.state);
        if (currentKey === JSON.stringify([1,2,3,4,5,6,7,8,0])) {
            // reconstruct path
            const path = [];
            let temp = current;
            while (temp) { path.unshift(temp); temp = temp.parent; }
            return path;
        }
        for (const move of getNeighbors(current.state)) {
            const nextKey = JSON.stringify(move.state);
```

```
            const newCost = costSoFar.get(currentKey) + 1;
            if (!costSoFar.has(nextKey) || newCost < costSoFar.get(nextKey)) {
                costSoFar.set(nextKey, newCost);
                const priority = newCost + manhattan(move.state);
                frontier.push({ state: move.state, parent: current, movedTile:
move.movedTile, direction: move.direction }, priority);
                cameFrom.set(nextKey, current);
            }
        }
    }
    return null;
}

function solveBacktracking(initialState) {
    const visited = new Set();
    const maxDepth = 35;
    let solutionPath = [];

    function dfs(node, depth) {
        if (depth > maxDepth) return false;
        const key = JSON.stringify(node.state);
        if (visited.has(key)) return false;
        visited.add(key);

        if (key === JSON.stringify([1,2,3,4,5,6,7,8,0])) {
            solutionPath = [node];
            return true;
        }
        for (const move of getNeighbors(node.state)) {
            const child = { state: move.state, parent: node, movedTile:
move.movedTile, direction: move.direction };
            if (dfs(child, depth + 1)) {
                solutionPath.unshift(node);
                return true;
            }
        }
        return false;
    }

    const root = { state: initialState, parent: null, movedTile: null,
direction: null };
    if (dfs(root, 0)) return solutionPath;
    return null;
}
```