8-Puzzle Solver Project

AI Project

Mohamed Ahmed Ramadan - 23015430
Mohamed Mahmoud Ibrahim - 23015446
Mazen Hussein Mostafa - 23017827

Table of Contents

1. Introduction

The 8-puzzle is a classic sliding puzzle problem that consists of a 3×3 grid with eight numbered tiles and one blank space. The goal is to rearrange the tiles from an initial configuration to a goal configuration by sliding tiles into the blank space. This project implements and compares three different search algorithms: A* Search, Breadth-First Search (BFS), and Depth-First Search (DFS).

The project features a graphical user interface (GUI) built with Tkinter that allows users to input custom puzzle configurations or select from preset examples. Users can solve puzzles using any of the three algorithms and visualize the solution path step-by-step. The implementation follows SOLID principles to ensure clean, maintainable, and extensible code architecture.

2. PEAS Description

2.1 Performance Measure

The performance of the puzzle solver is measured by:
• **Solution Optimality:** Number of moves required to reach the goal state (fewer is better)
• **Computational Efficiency:** Number of nodes explored during search (fewer is better)
• **Time Complexity:** Execution time to find the solution
• **Memory Usage:** Space required to store search frontier and visited states

2.2 Environment

Observable: Fully observable - the complete puzzle state is visible at all times
Deterministic: Each action produces a predictable outcome with no randomness
Static: The environment does not change unless the agent acts
Discrete: Finite number of states and actions
Single-agent: Only one agent solving the puzzle

2.3 Actuators

The agent can perform four possible actions by sliding tiles into the blank space:
• Move Up (slide tile below blank space upward)
• Move Down (slide tile above blank space downward)
• Move Left (slide tile right of blank space leftward)
• Move Right (slide tile left of blank space rightward)

2.4 Sensors

The agent perceives:
• Current puzzle configuration (3×3 grid state)
• Position of the blank space
• Whether current state matches the goal state

3. Problem Formulation

3.1 State Space

A state is represented as a 3×3 matrix of integers where 0 represents the blank space and 1-8 represent the numbered tiles. The state space consists of all possible configurations of the tiles. Theoretically, there are 9! = 362,880 possible permutations, but only half (181,440) are reachable from any given initial state due to parity constraints.

3.2 Initial State

The initial state can be any valid 3×3 configuration of numbers 0-8. The project allows users to input custom configurations or select from preset examples.

3.3 Goal State

The goal state is defined as:

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

where 0 is the blank space in the bottom-right corner.

3.4 Actions and Transition Model

Actions are defined by the valid moves of the blank space. From any state, 2 to 4 actions are possible depending on the blank position:
• Corner positions: 2 possible moves
• Edge positions: 3 possible moves
• Center position: 4 possible moves

The transition model produces a new state by swapping the blank space with an adjacent tile.

3.5 Path Cost

Each move has a uniform cost of 1. The path cost is the total number of moves from the initial state to the current state.

4. Search Algorithms

4.1 A* Search Algorithm

A* is an informed search algorithm that uses both the actual cost from the start (g) and an estimated cost to the goal (h) to guide the search. It maintains a priority queue ordered by f(n) = g(n) + h(n).

Heuristic Function: Manhattan Distance - the sum of horizontal and vertical distances each tile must move to reach its goal position. This heuristic is admissible (never overestimates) and consistent, guaranteeing optimal solutions.

Advantages:
• Guarantees optimal solution
• Generally explores fewer nodes than uninformed search
• Good balance between optimality and efficiency

Time Complexity: O(b^d) where b is branching factor, d is solution depth
Space Complexity: O(b^d) - stores all generated nodes

4.2 Breadth-First Search (BFS)

BFS is an uninformed search algorithm that explores all nodes at depth d before exploring nodes at depth d+1. It uses a FIFO queue to maintain the frontier.

Advantages:
• Guarantees optimal solution for uniform cost problems
• Complete - always finds a solution if one exists
• Simple to implement

Disadvantages:
• Explores many unnecessary nodes
• High memory consumption

Time Complexity: O(b^d)
Space Complexity: O(b^d)

4.3 Depth-First Search (DFS)

DFS is an uninformed search algorithm that explores as far as possible along each branch before backtracking. It uses a LIFO stack (or recursion) to maintain the frontier.

Implementation Detail: To prevent infinite loops, a maximum depth limit of 50 is imposed in this implementation.

Advantages:
• Low memory requirements
• May find solution quickly if lucky

Disadvantages:
• Does not guarantee optimal solution
• May get stuck in deep paths
• Not complete without depth limiting

Time Complexity: O(b^m) where m is maximum depth
Space Complexity: O(bm) - only stores path from root to current node

5. Results and Comparison

The three algorithms were tested on various puzzle configurations. Below is a comparison of their performance characteristics:

Criterion	A* Search	BFS	DFS
Optimality	Optimal	Optimal	Not Optimal
Completeness	Complete	Complete	Complete*
Nodes Explored	Low-Medium	High	Variable
Memory Usage	Medium	High	Low
Time Efficiency	Good	Poor	Variable
Best Use Case	General purpose	Guaranteed shortest	Memory constrained

**DFS completeness requires depth limiting to avoid infinite loops*

5.1 Example Performance Data

For a moderately difficult puzzle (10-15 moves to solution):

A* Search:
• Moves to solution: 12
• Nodes explored: ~50-100
• Execution time: <0.1 seconds

BFS:
• Moves to solution: 12 (optimal)
• Nodes explored: 500-2000
• Execution time: 0.2-0.5 seconds

DFS:
• Moves to solution: Variable (often 30-50, non-optimal)
• Nodes explored: 100-1000
• Execution time: Variable, may timeout at depth limit