Lodz University of Technology

Task 3

Fifteen Puzzle

Artificial Inteligence & Experts Systems
Tutor: MORAWSKA, Barbara

David Sánchez Sánchez
Lodz University of Technology
Poland, Lodz
905590@edu.p.lodz.pl

Alfonso Muñoz Sevillano
Lodz University of Technology
Poland, Lodz
905580@edu.p.lodz.pl

Index

1. Introduction

The N-Puzzle is a classical problem in artificial intelligence used to evaluate search strategies, heuristic design, and state-space exploration efficiency. In this project, we implemented a complete solver for the Fifteen Puzzle and its smaller variants, incorporating both uninformed and informed search algorithms. Our primary goal was to compare their behavior under different configurations and heuristic functions, as well as to provide a clean, modular, and fully documented implementation capable of solving arbitrary puzzle configurations from a text input.

The project is divided into two components:

1. Menu solver that reads the puzzle, runs the selected algorithm, and outputs the solution sequence
2. Visualization tool that displays the solution step by step.

Both components were developed with clarity, modularity, and academic reproducibility in mind.

2. Problem Definition

The Fifteen Puzzle consists of a 4×4 (can be diferent) grid with tiles numbered from 1 to 15 and a single empty space. A legal move shifts the empty space up, down, left, or right by swapping it with an adjacent tile. The objective is to transform an arbitrary initial configuration into a canonical goal configuration where tiles appear in increasing order, and the empty space occupies the bottom-right corner.

The search space is large: the 4×4 puzzle contains **16! / 2 ≈ 10 trillion reachable states**, which makes exhaustive enumeration impossible. As such, intelligent search algorithms and heuristics are essential to navigate this state space efficiently. The project investigates six search algorithms and multiple expansion modes and heuristics to evaluate their performance across puzzle sizes and difficulties.

3. Methodology

Our methodology follows a structured experimental approach:

1. **Implementation of all required search algorithms**, ensuring consistent interfaces:
    - Breadth-First Search (BFS)
    - Depth-First Search (DFS)
    - Iterative Deepening DFS (IDFS)
    - Greedy Best-First Search
    - A* Search
    - Simplified Memory-Bounded A* (SMA*)
2. **Definition of uniform successor-generation rules**, including:
    - Ordered expansions: *DULR*
    - Randomized expansion mode: *R*
3. **Integration of heuristic functions** for informed algorithms:
    - h_0: trivial (0)
    - h_1: misplaced tiles
    - h_2: Manhattan distance
4. **Execution of empirical tests** using two carefully selected inputs:
    - A simple 3×3 configuration solvable by all algorithms.
    - A difficult 4×4 configuration where uninformed algorithms fail, but informed algorithms with strong heuristics succeed.
5. **Develop a visualizer for solutions given**

6. **Extraction of quantitative results**, summarizing:
    - Whether a solution was found
    - Algorithm behavior under different modes
    - Observed patterns and insights

4. Design and Implementation

The project is implemented in Python, divided into multiple modules that separate concerns and promote readability. A placeholder for the GitHub repository link is provided below:

GitHub Repository:

https://github.com/Davicete7/FifteenPuzzle

HTML Documentation:

https://github.com/Davicete7/FifteenPuzzle/tree/main/codeDocumentation

The entire project is also accompanied by auto-generated HTML with **pydoc,** documentation containing detailed explanations for all classes, methods, and modules, ensuring full transparency and accessibility for external reviewers.

4.1 Key Components (High-Level Descriptions)

- **`puzzleState.py`**
  Represents a single configuration of the puzzle. It includes board storage, successor generation, cost tracking and path reconstruction.

- **`searchAlgorithms.py`**
  Contains clean implementations of all six algorithms required by the assignment. Each function returns:

- **`searchUtils.py`**
  Holds the goal-state generator and heuristic functions.

- **`main.py`**
  Implements input handling, the algorithm-selection menu, result printing, and a step-by-step visualization tool.

5. Experiments and Results

Experiments were carried out on two puzzle configurations of differing complexity. The tables below summarize each algorithm's performance depending on the expansion mode or heuristic.

5.1 Easy Input (3×3)

Input:

```
1 2 3
4 5 6
0 7 8
```

All algorithms successfully solved this configuration.

Algorithm	DULR	Random	h_0	h_1	h_2	Comment
1. BFS	✅	✅	—	—	—	Optimal solution found immediately
2. DFS	✅	✅	—	—	—	Works due to shallow depth
3. IDFS	✅	✅	—	—	—	Depth limit small (≤2)
4. Greedy (Best-First)	—	—	✅	✅	✅	All heuristics trivial for this state
5. A*	—	—	✅	✅	✅	Instant solution
6. SMA*	—	—	✅	✅	✅	Memory not a concern in 3×3

5.2 Hard Input (4×4)

Input:

```
5  1  2  6
13 9  3  4
14 11 8  7
10 15 12 0
```

In this significantly more complex puzzle, the algorithms behaved very differently:

Algorithm	DULR	Random	h_0	h_1	h_2	Comment
1. BFS	❌	❌	—	—	—	State space too large (memory exhaustion)
2. DFS	❌	❌	—	—	—	Gets lost in deep branches
3. IDFS	⚠️	⚠️	—	—	—	Depth > 25 makes it infeasible
4. Greedy	—	—	❌	⚠️	✅	Manhattan is usually sufficient
5. A*	—	—	⚠️	✅	✅	Manhattan yields fast, optimal solutions
6. SMA*	—	—	⚠️	✅	✅	Performs like A* depending on memory limit

6. Algoritms

6.1 Uninformed Search Algorithms

Breadth-First Search (BFS): Explores the frontier level by level, guaranteeing optimality. However, its memory usage grows exponentially, making it impractical for 4×4 puzzles despite being complete.

Depth-First Search (DFS): Expands the deepest unexplored node first. Efficient in memory but not complete: it may fall into infinite paths. In our implementation, expansion orders (*DULR* or *Random*) determine search priorities.

Iterative Deepening DFS (IDFS): Performs DFS with increasing depth limits (0, 1, 2, …). It is complete and optimal for unit-cost problems but becomes impractical when the solution depth exceeds ~20.

6.1.1 Expansion Modes

- **DULR:** Deterministic successor order: Down, Up, Left, Right.
  Ensures reproducible results.

- **Random Mode (R):** Successors are shuffled at each expansion, changing the search trajectory. Useful to escape unlucky deterministic patterns in DFS.

6.2 Informed Search Algorithms

Greedy Best-First Search: Expands the node with the smallest *h(n)*. Fast but not optimal. Highly dependent on heuristic accuracy.

A* Search: Uses $f(n) = g(n) + h(n)$. Optimal when *h* is admissible (h_1, h_2). Performs exceptionally well with Manhattan distance in 4×4 puzzles.

SMA*: A memory-bounded version of A*. If memory is exhausted, it discards the worst nodes but preserves enough information to reconstruct paths later. Practical for large puzzles but sensitive to memory limits.

6.2.1 Heuristics

- **h_0 (Trivial)**: Always 0 → reduces A* to uniform-cost search (i.e., BFS).

- **h_1 (Misplaced Tiles)**: Counts tiles not in goal position.

- **h_2 (Manhattan Distance)**: Sum of tile distances to their goal coordinates; the strongest and most consistent heuristic.

7. Comparing algoritms

By the test we have been than, this is our conclusion:

- **BFS and IDFS** excel on small puzzles but are infeasible for realistic 4×4 configurations, where the state space explodes beyond tractable limits.
- **DFS**, while memory-efficient, is unreliable for deep search spaces due to non-completeness and sensitivity to move ordering.
- **Greedy Best-First Search** becomes powerful only when paired with strong heuristics. Manhattan distance dramatically improves its performance, whereas h_0 and h_1 frequently misguide the search.
- **A*** consistently provides the best balance between completeness, optimality, and speed. With Manhattan distance, it reliably solves even difficult 4×4 configurations.
- **SMA*** mirrors A*'s behavior but introduces memory bounds, making it suitable for environments where memory is limited but search optimality is still desired.

Overall, **informed algorithms clearly outperform uninformed ones** on larger puzzles, and **the Manhattan distance heuristic is the decisive factor** in achieving practical solvability.

8. Conclusions

This project implemented and evaluated six classical search algorithms for solving the N-Puzzle, demonstrating both their theoretical properties and practical performance differences. Through a structured experimental approach, we observed the limitations of uninformed strategies on complex puzzles and the essential role of heuristic design in scaling to larger problem instances.

The Manhattan heuristic proved especially effective, enabling algorithms like A* and SMA* to solve configurations unreachable by BFS, DFS, or IDFS. The project also emphasized code modularity, documentation clarity, and the ability to visualize computed solutions, contributing to a complete and pedagogically robust exploration of search techniques in artificial intelligence.

The results highlight the importance of algorithmic selection and heuristic quality when tackling combinatorial search problems, reinforcing fundamental principles of AI search theory through empirical analysis.