CSCI 6511 – Artificial Intelligence
Project 1: Sub Project C – N-Puzzle

A* Search Solution

Ahmadov Kamal
Imamverdiyev Omar

February 9, 2026

1 Introduction

The N-Puzzle is a classic search problem in artificial intelligence that involves rearranging numbered tiles on an $n \times n$ grid into a predefined goal configuration using the minimum number of moves. One tile position is blank, and legal actions consist of sliding an adjacent tile horizontally or vertically into the blank space.

This project implements a solution to the N-Puzzle for grid sizes $3 \leq n \leq 8$ using the A* search algorithm. The objective is to find an optimal (minimum-length) sequence of moves that transforms a given initial configuration into the goal state.

2 Problem Description

An N-Puzzle instance consists of:

- An $n \times n$ grid containing tiles numbered from 1 to $n^2 - 1$
- One blank space represented by the value 0
- Legal moves that slide a tile into the blank position

The goal state is defined as:

$$(1, 2, 3, \ldots, n^2 - 1, 0)$$

with the blank tile located in the bottom-right corner.

The task is to find the shortest sequence of moves (up, down, left, right) that reaches the goal configuration.

3 Algorithm Choice: A* Search

To solve the N-Puzzle efficiently and optimally, we use the A* search algorithm. A* is an informed search method that expands nodes based on the evaluation function:

$$f(n) = g(n) + h(n)$$

where:

- $g(n)$ is the cost from the start state to the current state

- $h(n)$ is a heuristic estimate of the remaining cost to reach the goal

A* guarantees optimality provided that the heuristic function is admissible (never overestimates the true cost).

3.1 Heuristic Function

Our implementation uses a combination of two admissible heuristics:

- **Manhattan Distance**: the sum of the vertical and horizontal distances of each tile from its goal position.
- **Linear Conflict**: an enhancement that adds extra cost when two tiles are in the same row or column as their goal positions but in reversed order.

The final heuristic is:

$$h(n) = h_{\text{Manhattan}}(n) + h_{\text{Linear Conflict}}(n)$$

This heuristic remains admissible and significantly improves search performance.

4 Implementation Details

The program is implemented in Python and follows a command-line interface design.
Key components include:

- Robust input parsing supporting tab-delimited and space-aligned formats
- Solvability checking using inversion count rules
- A priority queue (min-heap) for A* frontier management
- State expansion with cost tracking
- Parent pointers for path reconstruction

A closed-set mechanism ensures that previously explored states with higher cost are not revisited.
No graphical user interface is used, in accordance with assignment guidelines.

4.1 Design Decisions

Several design choices were made to improve efficiency, correctness, and robustness:

- **Tuple-based board representation:** The puzzle board is represented as an immutable tuple, enabling fast hashing and efficient use as a key in explored-state and cost-tracking data structures.
- **Priority queue for frontier management:** A priority queue (min-heap) is used to manage the A* frontier, ensuring that states are expanded in order of increasing evaluation cost $f(n)$.
- **Parent pointers for path reconstruction:** Each explored state maintains a reference to its parent state and the corresponding move, allowing efficient reconstruction of the optimal solution path once the goal state is reached.

- **Robust input parsing:** The input parser supports instructor-provided files with tab-delimited, space-aligned, or missing blank entries, ensuring reliable parsing across different input formats.
- **Early solvability detection based on inversion parity:** A solvability test is performed before the A* search to avoid unnecessary computation. The test is based on permutation parity, computed by counting the number of inversions in the linearized board configuration while excluding the blank tile. For odd grid sizes, the puzzle is solvable if the inversion count is even, whereas for even grid sizes, solvability depends on the combined parity of the inversion count and the row position of the blank tile measured from the bottom. These conditions follow from a parity invariant preserved by all valid moves.

5 How to Run the Program

The program is executed from the command line using Python.

```
python npuzzle_astar.py input.txt
python npuzzle_astar.py input.txt --show
python npuzzle_astar.py input.txt --evaluation
python npuzzle_astar.py input.txt --evaluation --show
```

- `input.txt` contains the initial puzzle configuration
- The optional `--show` flag prints all intermediate board states along the solution path
- The optional `--evaluation` flag enables performance evaluation mode, in which Uniform Cost Search ($h = 0$) and A* search with heuristics are executed and compared

When both `--evaluation` and `--show` are specified, the program prints the full solution path *only* for the heuristic-guided A* search. The solution path for Uniform Cost Search is intentionally omitted, as it may involve a large number of intermediate states and does not provide additional insight beyond the reported performance metrics.
Note: The filename `input.txt` is used here as a generic example.

6 Experimental Results

For solvable configurations, the implemented algorithm produces the following outputs:

- The minimum number of moves required to reach the goal state
- The corresponding sequence of moves using the symbols U, D, L, and R
- Optionally, the complete sequence of intermediate board configurations

For unsolvable configurations, the program correctly detects the impossibility of reaching the goal state and reports that no solution exists.

6.1 Instructor-Provided 5×5 Puzzle

To validate the correctness and robustness of the implementation on larger problem instances, an instructor-provided 5×5 N-Puzzle configuration was tested. The initial state of the puzzle is shown below:

```
11  1  2  3 14
12  7  9 10 13
 6  8 18  5  4
21 16 17 19 15
22 23    24 20
```

This configuration was verified to be solvable using the parity-based solvability test. The algorithm successfully found an optimal solution for this instance.

6.2 Solution Output

The following results for the given 5×5 puzzle:

- **Minimum number of moves:** 38
- **Move sequence:** `LLURRULLUURRRDDRUULDDRUULDLLDLURRDRRDD`

The obtained solution demonstrates the ability of the A* algorithm with admissible heuristics to handle larger puzzle sizes beyond the standard 4×4 case. Despite the increased state space of the 5×5 puzzle, the algorithm efficiently guided the search toward the goal state and produced a valid optimal solution.

These experimental results confirm both the correctness of the implementation and its practical applicability to instructor-provided benchmark inputs.

7 Evaluation and Heuristic Comparison

To evaluate the impact of heuristic guidance in A* search, we compare the performance of A* with and without heuristics on a representative 4×4 (15-puzzle) instance. The goal of this experiment is to demonstrate how heuristics influence search efficiency, memory usage, and runtime, while maintaining solution optimality.

7.1 Compared Methods

The following two approaches are evaluated:

- **Uniform Cost Search (UCS):** Implemented as A* search with heuristic function $h(n) = 0$. In this case, state expansion is based solely on path cost, resulting in an uninformed search strategy.
- **A* with Heuristic:** A* search using an admissible heuristic composed of Manhattan distance combined with the linear conflict heuristic.

Both methods guarantee optimal solutions.

7.2 Evaluation Metrics

The comparison is based on the following metrics:

- Number of expanded states
- Maximum frontier size
- Runtime
- Optimal solution length

All experiments were conducted on the same initial configuration to ensure fairness.

7.3 Experimental Results

The evaluation was performed on the following 4×4 (15-puzzle) initial configuration:

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

The evaluated 4×4 puzzle required an optimal solution of 12 moves. Table 1 presents the measured performance of both search strategies.

Metric	**UCS (h = 0)**	**A* with Heuristic**
Expanded states	8,562	14
Maximum frontier size	7,909	13
Runtime (seconds)	0.072	< 0.001
Solution length	12	12

Table 1: Quantitative comparison of UCS and A* on a 4×4 puzzle

7.4 Discussion

Both Uniform Cost Search and A* search successfully produced the same optimal solution of length 12, confirming the correctness and fairness of the comparison. However, the computational effort required by the two methods differed substantially.

Uniform Cost Search expanded 8,562 states and maintained a frontier of up to 7,909 states, reflecting the exponential growth typical of uninformed search strategies. In contrast, A* search with heuristic guidance expanded only 14 states and required a maximum frontier size of 13 states. This corresponds to a reduction of more than two orders of magnitude in the explored search space.

The runtime measurements further illustrate this difference. While UCS required measurable execution time, the heuristic-guided A* search completed almost instantaneously. These results clearly demonstrate that admissible heuristics dramatically improve search efficiency while preserving optimality, even for moderate-sized N-Puzzle instances.

8 Limitations and Discussion

The N-Puzzle problem exhibits rapid growth in state space as the grid size n increases. For an $n \times n$ puzzle, the number of reachable configurations is on the order of $(n^2)!/2$, which quickly renders exhaustive search computationally infeasible for large values of n. Consequently, while A* search with admissible heuristics performs efficiently for small and moderate instances (such as 3×3, 4×4, and many 5×5 puzzles), larger configurations (6×6 to 8×8) may become impractical due to substantial increases in both computation time and memory usage.

A key limitation of A* search lies in its space complexity. To guarantee optimality, A* must store all generated states in both the OPEN (frontier) and CLOSED (explored) sets. In the worst case, the number of stored states grows exponentially with the depth of the optimal solution, often causing memory consumption to become the primary bottleneck before runtime alone becomes prohibitive.

The effectiveness of A* search is also strongly influenced by the quality of the heuristic function. Although the combination of Manhattan distance and linear conflict provides a tighter admissible estimate than Manhattan distance alone, it may still significantly underestimate the true remaining cost for deeply scrambled configurations. In such cases, many states share similar evaluation values $f(n) = g(n) + h(n)$, leading A* to expand a large number of nodes and gradually approach the behavior of Uniform Cost Search. More informed heuristics, such as pattern database heuristics, can further reduce node expansions but require additional preprocessing and increased memory resources.

Despite these limitations, A* remains an appropriate and optimal algorithm for solving the N-Puzzle within the given constraints. It guarantees minimal solution length while clearly illustrating the trade-offs between heuristic accuracy, computational efficiency, and memory usage, making it well suited for demonstrating key concepts of informed search in artificial intelligence.

9 Conclusion

This project demonstrates the application of A* search to solve the N-Puzzle optimally. By combining Manhattan distance with linear conflict heuristics, the implementation efficiently finds shortest solutions while adhering strictly to the project requirements. The solution illustrates key concepts of informed search and heuristic design in artificial intelligence.