CSE 318: Offline 1

Solve N-Puzzle

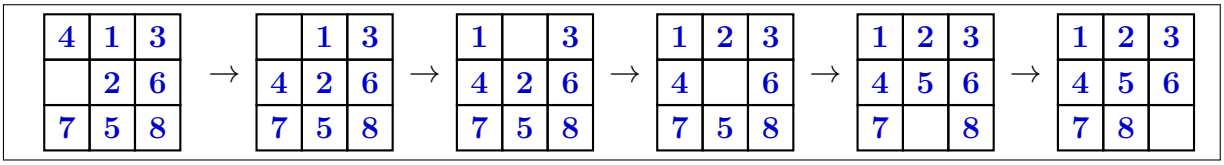
Objective: Write a program to solve the n-puzzle problem using the A* search algorithm, where

$$n = k^2 - 1, \quad k = 3, 4, 5, \ldots$$

Problem Description:

From the early days of artificial intelligence, the *n-puzzle* has served as a classic benchmark problem. It requires a degree of intelligent reasoning to find an optimal solution. The n-puzzle was invented and popularized in the 1870s by Noyes Palmer Chapman, a postmaster from Canastota, New York.

The game is played on a $k \times k$ grid containing $n = k^2 - 1$ square tiles labeled from 1 to n and a single blank square. The objective is to arrange the tiles in numerical order with the blank tile in the last position, using as few moves as possible. Players may slide any adjacent tile horizontally or vertically into the blank space, thereby changing the configuration of the board. The challenge lies in determining the shortest sequence of such legal moves that transforms an initial configuration into the goal configuration.



Solving the N-Puzzle Using A* Search

To solve the n-puzzle problem efficiently, we use the **A* search algorithm**, which finds the shortest sequence of moves from the initial configuration to the goal state.

In this approach, each *search node* represents

- The current board configuration
- A priority value
- A reference to the previous search node

The algorithm begins by inserting the **initial node** (initial board, 0 moves, and null previous node) into a **priority queue**, often called the *open list*.

The main loop of the algorithm works as follows:

1. Remove the node with the **lowest priority value** from the priority queue (this is the most promising node).

2. Add it to the *closed list*.

3. Generate all valid neighboring board configurations (reachable by one legal move).

4. For each neighbor not in the closed list, insert it into the priority queue.

This process repeats until the goal board is dequeued from the priority queue. At that point, the algorithm has found an optimal solution.

The key component of A* is its **priority function**:

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

Where:

- $g(n)$: The cost to reach node n from the start (i.e., number of moves made so far)
- $h(n)$: A heuristic estimate of the cost to reach the goal from node n

The heuristic $h(n)$ plays a crucial role. A better heuristic leads to fewer nodes being explored and a faster solution. The choice of heuristic affects the efficiency of the algorithm but not the correctness — as long as $h(n)$ is *admissible* (i.e., never overestimates the true cost), A* is guaranteed to find the optimal solution.

Heuristic Functions

To guide the A* search algorithm effectively, we use heuristic functions that estimate the cost from a given state to the goal state. The better (more accurate and admissible) the heuristic, the fewer nodes the algorithm needs to explore.

1. **Hamming Distance**: Counts the number of tiles that are not in their goal position. The blank tile is not included in this count.

2. **Manhattan Distance**: Calculates the sum of the vertical and horizontal distances each tile must move to reach its goal position. This is one of the most commonly used admissible heuristics for the n-puzzle.

3. **Euclidean Distance**: Computes the straight-line distance from each tile's current position to its goal position, based on the Euclidean formula:

$$\sqrt{(x_{\text{current}} - x_{\text{goal}})^2 + (y_{\text{current}} - y_{\text{goal}})^2}$$

4. **Linear Conflict**: A more informed heuristic that enhances Manhattan distance by considering pairs of tiles that are in the correct row or column but in the wrong order.

   Two tiles t_j and t_k are said to be in *linear conflict* if:

   - They are in the same line (row or column),
   - Their goal positions are also in that line,
   - Tile t_j is to the right (or below) of t_k in the current state,
   - But the goal position of t_j is to the left (or above) the goal position of t_k.

In such cases, at least one of the tiles must move out of the way to allow the other to pass — resulting in at least two additional moves. The linear conflict heuristic is calculated as:

$$h(n) = \text{Manhattan Distance} + 2 \times (\text{Number of Linear Conflicts})$$

Let us assume that our initial board is

7	**2**	**4**
6		**5**
8	**3**	**1**

Our goal board is

1	**2**	**3**
4	**5**	**6**
7	**8**	

From the following table we can calculate Hamming distance = **7**

Digits	1	2	3	4	5	6	7	8
Is placed correctly (1 = no, 0 = yes)	1	0	1	1	1	1	1	1

From the following table we can calculate Manhattan distance = **16**

Digits	1	2	3	4	5	6	7	8
Row distance	2	0	2	1	0	0	2	0
Column distance	2	0	1	2	1	2	0	1
Total distance	4	0	3	3	1	2	2	1

From the following table we can calculate Euclidean distance = **13.30**

Digits	1	2	3	4	5	6	7	8
Distance	$\sqrt{8}$	0	$\sqrt{5}$	$\sqrt{5}$	1	2	2	1
Value (approx)	2.828	0	2.236	2.236	1	2	2	1

Linear conflict count = **1 (in 2nd row)**
So, Linear Conflict heuristic = **Manhattan distance + 2 × 1 = 18**

Detecting Unsolvable Puzzles

Not all initial board configurations of the n-puzzle can be solved. That is, not every arrangement of tiles can be transformed into the goal state using legal moves. For example, the following board is unsolvable:

8	1	2
	4	3
7	6	5

This happens because the set of all possible board states can be divided into two **equivalence classes** — those that can reach the goal state, and those that cannot. Whether a board belongs to the solvable or unsolvable class can be determined by computing the number of **inversions** in the board.

Inversion: An inversion is a pair of tiles (i, j) such that $i < j$ but tile i appears *after* tile j in the board's row-major ordering (i.e., reading rows left to right, top to bottom). The blank tile is ignored.

1	2	3
	4	6
8	5	7

row-major order: 1 2 3 4 6 8 5 7
3 inversions: 6-5, 8-5, 8-7

	1	3
4	2	5
7	8	6

row-major order: 1 3 4 2 5 7 8 6
4 inversions: 3-2, 4-2, 7-6, 8-6

The rules for solvability depend on the size of the grid:

- **If k is odd** (e.g., 3×3 grid): The puzzle is solvable if the total number of inversions is **even**.
- **If k is even** (e.g., 4×4 grid): Let the row of the blank tile be counted from the *bottom* (starting from 1). The puzzle is solvable if:
  - The blank is on an even row from the bottom **and** the number of inversions is odd, or
  - The blank is on an odd row from the bottom **and** the number of inversions is even.

  For all other cases, the puzzle is unsolvable.

Additional Resources

- **Is the given puzzle solvable? (1 to 9)**
- **Is the given puzzle solvable? (1 to 15)**
- **Visualizing N-Puzzle**

Tasks

1. Implement the A* search algorithm using the heuristic algorithms. Your implementation should allow the user to easily switch between heuristics without modifying the core A* algorithm. Aim for modular, extensible code where heuristic logic is separated from the search logic. Your program should take the following inputs:
   - Grid size k
   - Initial board configuration (use 0 to represent the blank tile)
2. Determine whether the input board configuration is solvable. If it is solvable:
   (a) Output the optimal cost to reach the goal state and the sequence of board configurations representing each step.
   (b) Additionally, print the number of nodes **explored** and **expanded**.

Input and output formats

The input and output format for a board is the board size N followed by the N-by-N initial board, using 0 to represent the blank square.

Input	Output
3 0 1 3 4 2 5 7 8 6	Minimum number of moves = 4 0 1 3 4 2 5 7 8 6 1 0 3 4 2 5 7 8 6 1 2 3 4 0 5 7 8 6 1 2 3 4 5 0 7 8 6 1 2 3 4 5 6 7 8 0
3 1 2 3 0 4 6 7 5 8	Minimum number of moves = 3 1 2 3 0 4 6 7 5 8 1 2 3 4 0 6 7 5 8 1 2 3 4 5 6 7 0 8 1 2 3 4 5 6 7 8 0

3 1 2 3 4 5 6 8 7 0	Unsolvable puzzle

Submission Deadline

28 April, Monday, 11:55 PM