A Search Implementation for the 8-Puzzle Problem*

8-Puzzle Problem Formulation

The 8-puzzle is a classic sliding block puzzle that can be framed as a formal search problem.

- **States**: A state is a unique 3x3 grid configuration of the numbers 1-8 and a single empty space (represented by 0).
- **Initial State**: The starting configuration of the grid provided by the user.
- **Goal State**: The target configuration that the search aims to reach.
- **Operators (Actions)**: The four actions that transition between states: moving the empty space **Up**, **Down**, **Left**, or **Right**. An action is only valid if the empty space remains within the 3x3 grid.
- **Path Cost**: The cost of a solution is the number of moves taken. Each move has a uniform cost of 1.

Program Structure

The program is structured around the A* search algorithm and uses specific data structures for efficiency.

- **Core Data Structure**: A `PuzzleNode` class (or struct) is central to the program. Each `PuzzleNode` object stores:
  - `state`: The current 3x3 grid configuration.
  - `parent`: A reference to the node that generated this one, used for path reconstruction.
  - `action`: The move (`'Up'`, `'Down'`, etc.) that led to this state.
  - `g`: The cost from the start node to the current node (path depth).
  - `f`: The total estimated cost, calculated as $f=g+h$.
- **Control Structures**:
  - **Open List (Frontier)**: A **priority queue** (implemented as a min-heap) stores nodes that are yet to be explored. Nodes are ordered by their `f-cost`, ensuring the algorithm always expands the most promising node first.
  - **Closed List (Explored Set)**: A **hash set** stores the states that have already been visited. This is crucial for preventing cycles and redundant computations, providing fast O(1) average time complexity for lookups.

Global Variables and Key Data

While true global variables are avoided, the main execution scope relies on several key variables.

- `initial_state`: A 3x3 list or array holding the starting configuration provided by the user.
- `goal_state`: A 3x3 list or array holding the target configuration.
- `open_list`: The priority queue data structure described above.
- `closed_set`: The hash set data structure described above.

Functions and Procedures

The program is modularized into several key functions, each with a specific responsibility.

- `main()`:
  - **Purpose**: Serves as the program's entry point.
  - **Responsibilities**: Handles user input for the initial and goal states, calls the `is_solvable` check, initiates the `a_star_search` function, and prints the final path or failure message.
- `a_star_search(initial_state, goal_state, heuristic_func)`:
  - **Purpose**: Implements the core A* algorithm.
  - **Responsibilities**: Initializes the start node, manages the `open_list` and `closed_set`, loops through nodes, expands the best one, and calls `reconstruct_path` upon finding the goal.
- `get_successors(node)`:
  - **Purpose**: Generates all valid child nodes from a given parent node.
  - **Responsibilities**: Finds the location of the empty space (0), determines which of the four moves are legal, and creates a new `PuzzleNode` for each valid successor state.
- `heuristic_func(state, goal_state)`:
  - **Purpose**: A placeholder for the specific heuristic calculation. Two versions are implemented:
    1. `misplaced_tiles()`: Counts tiles not in their goal position.
    2. `manhattan_distance()`: Sums the distances of each tile from its goal position.
  - **Responsibilities**: Returns the estimated cost `h` from the current state to the goal.
- `is_solvable(state)`:
  - **Purpose**: To determine if a puzzle configuration has a solution before starting the search.

    - **Responsibilities**: Calculates the number of **inversions** in the initial state. Returns `True` if the count is even (solvable) and `False` if it's odd (unsolvable).
- `reconstruct_path(goal_node)`:
    - **Purpose**: To build the final solution path after the goal is found.
    - **Responsibilities**: Traverses backwards from the `goal_node` to the start node using the `parent` reference in each node, accumulating the states and actions along the way.

Sample input and output

```
C:\Users\jeevi\OneDrive\Desktop\pyhton code for AI>python eight_puzzle.py
Enter the initial state (3x3 grid, use 0 for empty space):
1 2 3
4 5 0
7 8 6

Enter the goal state (3x3 grid, use 0 for empty space):
1 2 3
4 5 6
7 8 0

--- Solving with h1: Misplaced Tiles Heuristic ---
Path found in 1 moves.
Nodes Generated: 4
Nodes Expanded: 1

--- Solving with h2: Manhattan Distance Heuristic ---
Path found in 1 moves.
Nodes Generated: 4
Nodes Expanded: 1
```

Results table

Case	Heuristic method	Solution length	Nodes generated	Nodes expanded
1	Misplaced Tiles	1	4	1
1	Manhattan Distance	1	4	1
2	Misplaced Tiles	2	8	2
2	Manhattan Distance	2	8	2
3	Misplaced Tiles	8	51	18
3	Manhattan Distance	8	35	12
4	Misplaced Tiles	24	48634	17987
4	Manhattan Distance	24	4740	1773
5	Misplaced Tiles	23	29023	10681
5	Manhattan Distance	23	2077	778
6	Misplaced Tiles	22	17589	6488
6	Manhattan Distance	22	891	329
7	Misplaced Tiles	20	8855	3281
7	Manhattan Distance	20	518	193
8	Misplaced Tiles	28	170064	63097
8	Manhattan Distance	28	4304	1622