DEPARTMENT OF CSE(AI & AIML)
Introduction to AI
Report File
8-Puzzle Solver

Name: Manu Garg

Univ roll no. : 202401100300150

branch : CSE(AI)

section: C

date of submmission: 11/03/2025

8-Puzzle Solver
Introduction to AI

1. Introduction:

The 8-puzzle is a classic sliding tile puzzle that consists of a 3×3 grid containing 8 numbered tiles (1-8) and one empty space. The objective is to rearrange the tiles from an initial configuration to a specified goal state by sliding tiles into the adjacent empty space. This puzzle serves as an excellent platform for demonstrating search algorithms and heuristic functions in artificial intelligence.

This report documents the implementation of an 8-puzzle solver using the A* search algorithm with Manhattan distance heuristic. The A* algorithm is particularly well-suited for this problem as it guarantees an optimal solution (fewest moves) when used with an admissible heuristic like Manhattan distance.

Key features of this implementation include:

- A* search algorithm for optimal path finding

- Manhattan distance heuristic for efficient search

- Solvability checking to determine if a solution exists

- Step-by-step solution display

- User input functionality for custom puzzle configurations

2. Methodology

2.1 Problem Representation

Each state of the 8-puzzle is represented as a 3×3 grid. In our implementation, we use a 2D list to represent the board, with 0 representing the empty space. Each puzzle state includes:

- The current board configuration

- Number of moves made to reach this state

- Reference to the previous state (for path reconstruction)

- Position of the empty space

- A hash value for efficient state comparison

2.2 A* Search Algorithm

The A* search algorithm uses a priority queue to explore states in order of their estimated total cost:

- g(n): The cost to reach the current state (number of moves made)

- h(n): The estimated cost to reach the goal (Manhattan distance heuristic)

- f(n) = g(n) + h(n): The total estimated cost

The algorithm maintains two sets:

- Open set: States that have been discovered but not yet explored
- Closed set: States that have already been explored

By always exploring the state with the lowest f(n) value, A* efficiently finds the optimal solution.

2.3 Manhattan Distance Heuristic

The Manhattan distance heuristic calculates the sum of the horizontal and vertical distances each tile must move to reach its goal position. This provides an admissible heuristic (never overestimates the actual cost), ensuring A* finds an optimal solution.

2.4 Solvability Check

Not all initial configurations of the 8-puzzle are solvable. A configuration is solvable if and only if the number of inversions (pairs of tiles that are in the wrong order relative to each other) is even. Our implementation includes a solvability check to determine if a solution exists before attempting to solve the puzzle.

3. Implementation

The implementation consists of a `PuzzleState` class for representing puzzle states and functions for solving and displaying the solution.

```python
import heapq
import copy
from collections import deque


class PuzzleState:
    def __init__(self, board, moves=0, previous=None):
        self.board = board
        self.moves = moves
        self.previous = previous
        self.empty_pos = self.find_empty()
        self.hash_value = None


    def find_empty(self):
        """Find the position of the empty space (0)"""
        for i in range(3):
            for j in range(3):
                if self.board[i][j] == 0:
                    return (i, j)
        return (-1, -1)  # Should never happen
```

```
    def get_neighbors(self):
        """Generate all possible next states"""
        neighbors = []
        i, j = self.empty_pos
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # Right, Down, Left, Up

        for di, dj in directions:
            ni, nj = i + di, j + dj
            if 0 <= ni < 3 and 0 <= nj < 3:
                # Create a new board with the tile moved
                new_board = copy.deepcopy(self.board)
                new_board[i][j] = new_board[ni][nj]
                new_board[ni][nj] = 0
                neighbors.append(PuzzleState(new_board, self.moves + 1, self))

        return neighbors

    def manhattan_distance(self, goal):
        """Calculate Manhattan distance heuristic"""
        distance = 0
        for i in range(3):
            for j in range(3):
```

```
                if self.board[i][j] != 0:
                    # Find where this tile should be in the goal state
                    for gi in range(3):
                        for gj in range(3):
                            if goal.board[gi][gj] == self.board[i][j]:
                                distance += abs(i - gi) + abs(j - gj)
        return distance

    def __lt__(self, other):
        # Required for heapq to compare PuzzleState objects
        return (self.moves + self.manhattan_distance(goal_state)) <
(other.moves + other.manhattan_distance(goal_state))

    def __eq__(self, other):
        return self.board == other.board

    def __hash__(self):
        if self.hash_value is None:
            self.hash_value = hash(str(self.board))
        return self.hash_value

    def is_solvable(self):
        """Check if the puzzle is solvable"""
```

```
        # Flatten the board
        flat_board = [num for row in self.board for num in row if num != 0]

        # Count inversions
        inversions = 0
        for i in range(len(flat_board)):
            for j in range(i + 1, len(flat_board)):
                if flat_board[i] > flat_board[j]:
                    inversions += 1

        return inversions % 2 == 0

    def __str__(self):
        result = ""
        for row in self.board:
            result += " ".join(str(tile) if tile != 0 else "_" for tile in row) + "\n"
        return result

def solve_puzzle(initial_state, goal_state):
    """Solve the 8-puzzle using A* search algorithm"""
    if not initial_state.is_solvable():
        return None, "This puzzle is not solvable."
```

```
    open_set = []
    closed_set = set()

    # Add initial state to open set
    heapq.heappush(open_set, initial_state)

    nodes_expanded = 0

    while open_set:
        current = heapq.heappop(open_set)
        nodes_expanded += 1

        # Check if we reached the goal
        if current.board == goal_state.board:
            # Reconstruct path
            path = []
            while current:
                path.append(current)
                current = current.previous
            return list(reversed(path)), f"Solved in {len(path) - 1} moves.
Expanded {nodes_expanded} nodes."

        # Add current state to closed set
```

```python
        closed_set.add(hash(str(current.board)))

        # Explore neighbors
        for neighbor in current.get_neighbors():
            if hash(str(neighbor.board)) not in closed_set:
                heapq.heappush(open_set, neighbor)

    return None, "No solution found."

def print_solution(solution):
    """Print the solution path"""
    if solution:
        for i, state in enumerate(solution):
            print(f"Step {i}:")
            print(state)

# Example usage
def get_user_input():
    """Get puzzle configuration from user"""
    print("Enter your 8-puzzle configuration (use 0 for empty space):")
    print("Format: Enter 3 rows of 3 numbers each, separated by spaces")
    board = []
    for i in range(3):
```

```
        while True:
            try:
                row = list(map(int, input(f"Row {i+1}: ").strip().split()))
                if len(row) != 3 or not all(0 <= num <= 8 for num in row):
                    print("Invalid input. Please enter 3 numbers between 0-8.")
                    continue
                board.append(row)
                break
            except ValueError:
                print("Invalid input. Please enter numbers only.")
    return board


def main():
    # Either get user input or use default example
    use_default = input("Use default puzzle? (y/n): ").lower() == 'y'

    if use_default:
        initial_board = [
            [1, 2, 3],
            [4, 0, 6],
            [7, 5, 8]
        ]
    else:
```

```
        initial_board = get_user_input()

    global goal_state
    goal_board = [
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 0]
    ]

    initial_state = PuzzleState(initial_board)
    goal_state = PuzzleState(goal_board)

    print("\nInitial State:")
    print(initial_state)

    print("Goal State:")
    print(goal_state)

    print("\nSolving...")
    solution, message = solve_puzzle(initial_state, goal_state)

    print(message)
    if solution:
```

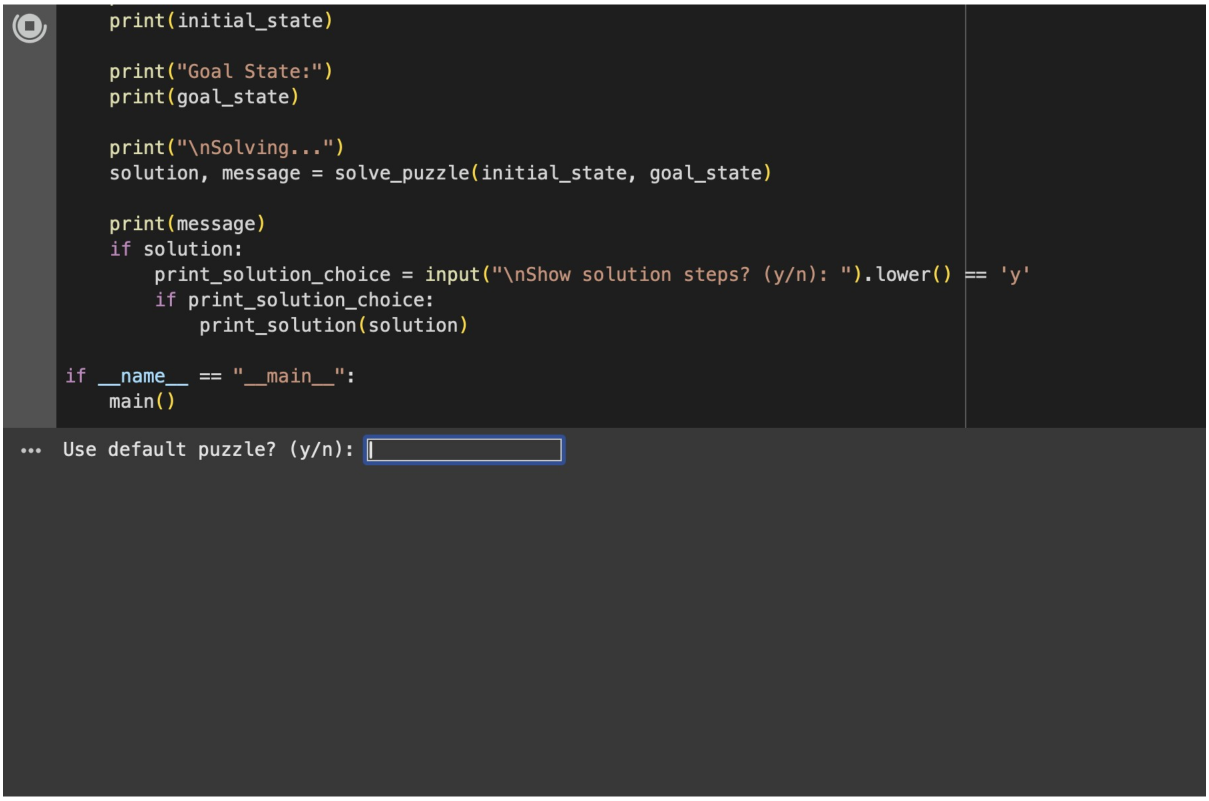
```
        print_solution_choice = input("\nShow solution steps? (y/n): ").lower() == 'y'
        if print_solution_choice:
            print_solution(solution)


if __name__ == "__main__":
    main()
```

4. Results and Output Screenshots

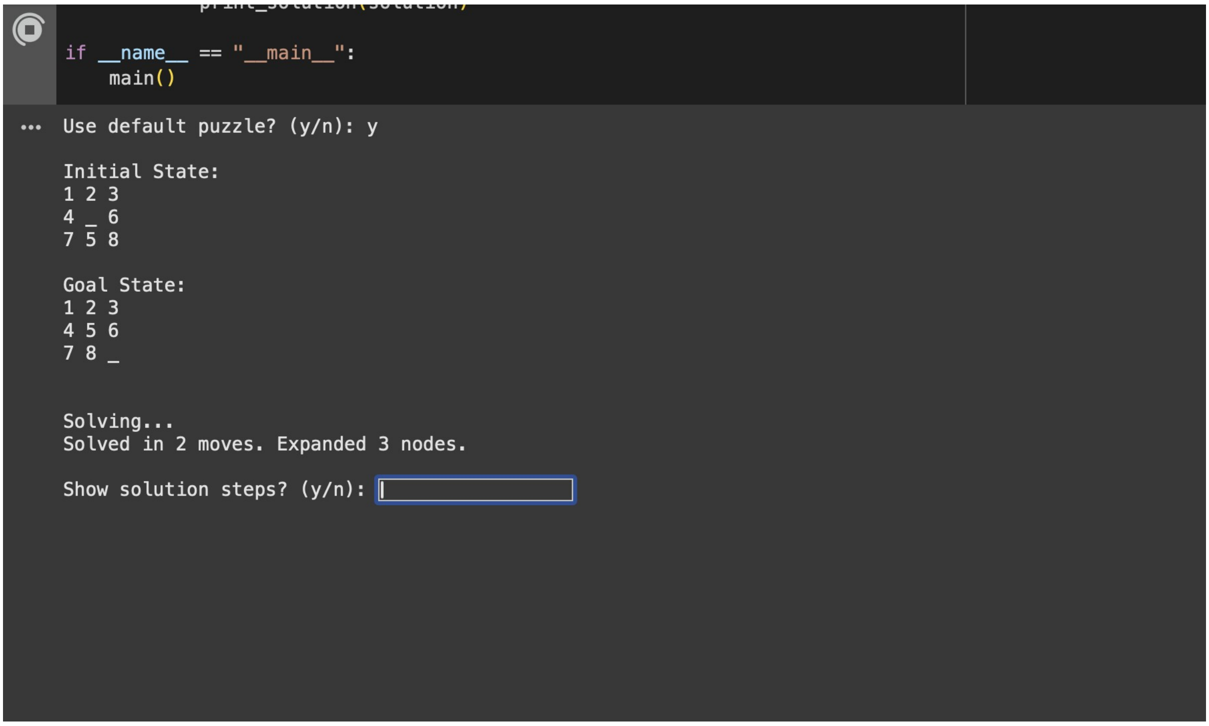
Below are screenshots showing the execution of the 8-puzzle solver:

[Screenshot 1: Program startup showing initial prompt]



Description: Screenshot showing the initial prompt asking the user whether to use the default puzzle or input a custom one.

[Screenshot 2: Example of solving default puzzle]



Description: Screenshot showing the program solving the default puzzle, displaying the initial state, goal state, and solution information.

[Screenshot 3: Solution steps]

```
Use default puzzle? (y/n): y

Initial State:
1 2 3
4 _ 6
7 5 8

Goal State:
1 2 3
4 5 6
7 8 _


Solving...
Solved in 2 moves. Expanded 3 nodes.

Show solution steps? (y/n): y
Step 0:
1 2 3
4 _ 6
7 5 8

Step 1:
1 2 3
4 5 6
7 _ 8

Step 2:
1 2 3
4 5 6
7 8 _
```

Description: Screenshot showing the step-by-step solution from the initial state to the goal state.

[Screenshot 4: Solvability check example]

```
        if print_solution_choice:
            print_solution(solution)

if __name__ == "__main__":
    main()
```

```
Use default puzzle? (y/n): n
Enter your 8-puzzle configuration (use 0 for empty space):
Format: Enter 3 rows of 3 numbers each, separated by spaces
Row 1: 1 2 3
Row 2: 4 5 6
Row 3: 8 7 0

Initial State:
1 2 3
4 5 6
8 7 _

Goal State:
1 2 3
4 5 6
7 8 _


Solving...
This puzzle is not solvable.
```

Description: Screenshot demonstrating the program's ability to identify an unsolvable puzzle configuration.

[Screenshot 5: Custom puzzle input]

```
            print_solution(solution)

if __name__ == "__main__":
    main()
```

```
Use default puzzle? (y/n): n
Enter your 8-puzzle configuration (use 0 for empty space):
Format: Enter 3 rows of 3 numbers each, separated by spaces
Row 1: 1 3 0
Row 2: 4 2 6
Row 3: 7 5 8

Initial State:
1 3 _
4 2 6
7 5 8

Goal State:
1 2 3
4 5 6
7 8 _


Solving...
Solved in 4 moves. Expanded 5 nodes.

Show solution steps? (y/n): y
Step 0:
1 3 _
4 2 6
7 5 8
```

Description: Screenshot showing the user inputting a custom puzzle configuration.