1BM22CS235

Lab-3b

8 Puzzle hueristic

1. Using Misplaced Tiles approach

```
import numpy as np

import heapq


class PuzzleState:

    def __init__(self, board, level, goal):

        self.board = board

        self.level = level

        self.goal = goal

        self.blank_pos = self.find_blank()

        self.cost = self.level + self.misplaced_tiles()


    def find_blank(self):

        return tuple(np.argwhere(self.board == 0)[0])


    def misplaced_tiles(self):

        return np.sum(self.board != self.goal) - (self.board[self.board == 0] != self.goal[self.goal ==
0]).sum()


    def get_neighbors(self):

        neighbors = []

        x, y = self.blank_pos

        moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right
```

```
        move_names = ['Up', 'Down', 'Left', 'Right']

        for (dx, dy), move_name in zip(moves, move_names):
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_board = self.board.copy()
                new_board[x, y], new_board[new_x, new_y] = new_board[new_x, new_y],
new_board[x, y]
                neighbors.append((PuzzleState(new_board, self.level + 1, self.goal), move_name))

        return neighbors

    def __lt__(self, other):
        return self.cost < other.cost

def print_board(state):
    print("\nCurrent State:")
    print(state.board)
    print(f"Misplaced Tiles: {state.misplaced_tiles()}\n")

def a_star(initial_state, goal_state):
    open_set = []
    closed_set = set()
    heapq.heappush(open_set, initial_state)

    print("Initial State:")
    print_board(initial_state)

    while open_set:
        current = heapq.heappop(open_set)
```

```
        if np.array_equal(current.board, current.goal):
            print("Goal state reached!")
            print(f"Total Cost: {current.cost}")
            print_board(current)
            return

        closed_set.add(tuple(map(tuple, current.board)))

        neighbors = current.get_neighbors()
        best_neighbor = None

        for neighbor, move_name in neighbors:
            if tuple(map(tuple, neighbor.board)) in closed_set:
                continue

            if best_neighbor is None or neighbor < best_neighbor[0]:
                best_neighbor = (neighbor, move_name)

        if best_neighbor:
            print(f"Moved {best_neighbor[1]}")
            print_board(best_neighbor[0])
            heapq.heappush(open_set, best_neighbor[0])

def main():
    print("Enter the initial state (3x3) as a single line of numbers (0 for blank):")
    initial_state_input = list(map(int, input().split()))
    initial_state = np.array(initial_state_input).reshape(3, 3)

    print("Enter the goal state (3x3) as a single line of numbers (0 for blank):")
    goal_state_input = list(map(int, input().split()))
    goal_state = np.array(goal_state_input).reshape(3, 3)
```

```
    initial_puzzle = PuzzleState(initial_state, 0, goal_state)
    a_star(initial_puzzle, goal_state)

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

<u>Output</u>

```
1 2 3 8 0 4 7 6 5
Initial State:

Current State:
[[2 8 3]
 [1 6 4]
 [7 0 5]]
Misplaced Tiles: 5

Moved Up

Current State:
[[2 8 3]
 [1 0 4]
 [7 6 5]]
Misplaced Tiles: 3

Moved Up

Current State:
[[2 0 3]
 [1 8 4]
 [7 6 5]]
Misplaced Tiles: 4
```

```
Moved Left

Current State:
[[0 2 3]
 [1 8 4]
 [7 6 5]]
Misplaced Tiles: 3

Moved Down

Current State:
[[1 2 3]
 [0 8 4]
 [7 6 5]]
Misplaced Tiles: 2

Moved Right

Current State:
[[1 2 3]
 [8 0 4]
 [7 6 5]]
Misplaced Tiles: 0

Goal state reached!
Total Cost: 5

Current State:
[[1 2 3]
 [8 0 4]
 [7 6 5]]
Misplaced Tiles: 0
```

2. Using Manhattan Distance approach

```
import numpy as np

import heapq


class PuzzleState:

    def __init__(self, board, level, goal):

        self.board = board

        self.level = level

        self.goal = goal

        self.blank_pos = self.find_blank()

        self.cost = self.level + self.manhattan_distance()


    def find_blank(self):

        return tuple(np.argwhere(self.board == 0)[0])


    def manhattan_distance(self):

        distance = 0

        goal_positions = {value: (i, j) for i, row in enumerate(self.goal) for j, value in
enumerate(row)}

        for i in range(3):

            for j in range(3):

                value = self.board[i, j]

                if value != 0:

                    goal_x, goal_y = goal_positions[value]

                    distance += abs(goal_x - i) + abs(goal_y - j)

        return distance


    def get_neighbors(self):

        neighbors = []

        x, y = self.blank_pos
```

```
        moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        move_names = ['Up', 'Down', 'Left', 'Right']

        for (dx, dy), move_name in zip(moves, move_names):
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_board = self.board.copy()
                new_board[x, y], new_board[new_x, new_y] = new_board[new_x, new_y],
new_board[x, y]
                neighbors.append((PuzzleState(new_board, self.level + 1, self.goal), move_name))

        return neighbors

    def __lt__(self, other):
        return self.cost < other.cost

def print_board(state):
    print("\nCurrent State:")
    print(state.board)
    print(f"Manhattan Distance: {state.manhattan_distance()}\n")

def a_star(initial_state, goal_state):
    open_set = []
    closed_set = set()
    heapq.heappush(open_set, initial_state)

    print("Initial State:")
    print_board(initial_state)

    while open_set:
        current = heapq.heappop(open_set)
```

```
        if np.array_equal(current.board, current.goal):
            print("Goal state reached!")
            print(f"Total Cost: {current.cost}")
            print_board(current)
            return

        closed_set.add(tuple(map(tuple, current.board)))

        neighbors = current.get_neighbors()
        best_neighbor = None

        for neighbor, move_name in neighbors:
            if tuple(map(tuple, neighbor.board)) in closed_set:
                continue

            if best_neighbor is None or neighbor < best_neighbor[0]:
                best_neighbor = (neighbor, move_name)

        if best_neighbor:
            print(f"Moved {best_neighbor[1]}")
            print_board(best_neighbor[0])
            heapq.heappush(open_set, best_neighbor[0])

def main():
    print("Enter the initial state (3x3) as a single line of numbers (0 for blank):")
    initial_state_input = list(map(int, input().split()))
    initial_state = np.array(initial_state_input).reshape(3, 3)

    print("Enter the goal state (3x3) as a single line of numbers (0 for blank):")
    goal_state_input = list(map(int, input().split()))
```

```python
    goal_state = np.array(goal_state_input).reshape(3, 3)

    initial_puzzle = PuzzleState(initial_state, 0, goal_state)
    a_star(initial_puzzle, goal_state)

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

Output:

```
Enter the initial state (3x3) as a single line of numbers (0 for blank):
2 8 3 1 6 4 7 0 5
Enter the goal state (3x3) as a single line of numbers (0 for blank):
1 2 3 8 0 4 7 6 5
Initial State:

Current State:
[[2 8 3]
 [1 6 4]
 [7 0 5]]
Manhattan Distance: 5

Moved Up

Current State:
[[2 8 3]
 [1 0 4]
 [7 6 5]]
Manhattan Distance: 4

Moved Up

Current State:
[[2 0 3]
 [1 8 4]
 [7 6 5]]
Manhattan Distance: 3
```

```
Current State:
[[0 2 3]
 [1 8 4]
 [7 6 5]]
Manhattan Distance: 2

Moved Down

Current State:
[[1 2 3]
 [0 8 4]
 [7 6 5]]
Manhattan Distance: 1

Moved Right

Current State:
[[1 2 3]
 [8 0 4]
 [7 6 5]]
Manhattan Distance: 0

Goal state reached!
Total Cost: 5

Current State:
[[1 2 3]
 [8 0 4]
 [7 6 5]]
Manhattan Distance: 0
```