LAB 4: A* Search

Manhattan Distance Heuristic

```
import heapq
import time

class PuzzleState:
    def __init__(self, board, goal, path="", cost=0):
        self.board = board
        self.goal = goal
        self.path = path
        self.cost = cost
        self.zero_pos = self.board.index(0)
        self.size = int(len(board) ** 0.5)

    def __lt__(self, other):

        return (self.cost + self.heuristic()) < (other.cost + other.heuristic())

    def heuristic(self):

        distance = 0
        for i, tile in enumerate(self.board):
            if tile != 0:
                goal_pos = self.goal.index(tile)
                distance += abs(i // self.size - goal_pos // self.size) + abs(i % self.size - goal_pos %
self.size)
        return distance

    def get_neighbors(self):
        neighbors = []
        x, y = divmod(self.zero_pos, self.size)
        moves = {'U': (x - 1, y), 'D': (x + 1, y), 'L': (x, y - 1), 'R': (x, y + 1)}

        for move, (nx, ny) in moves.items():
            if 0 <= nx < self.size and 0 <= ny < self.size:
                new_zero_pos = nx * self.size + ny
                new_board = list(self.board)

                new_board[self.zero_pos], new_board[new_zero_pos] = new_board[new_zero_pos],
new_board[self.zero_pos]
                neighbors.append(PuzzleState(tuple(new_board), self.goal, self.path + move,
self.cost + 1))
```

```python
        return neighbors

def a_star(start, goal):
    start_state = PuzzleState(start, goal)
    frontier = []
    heapq.heappush(frontier, start_state)
    explored = set()
    parent_map = {start_state.board: None}
    move_map = {start_state.board: ""}

    while frontier:
        current_state = heapq.heappop(frontier)

        if current_state.board == goal:
            return reconstruct_path(parent_map, move_map, current_state.board)

        explored.add(current_state.board)

        for neighbor in current_state.get_neighbors():
            if neighbor.board not in explored and neighbor.board not in parent_map:
                parent_map[neighbor.board] = current_state.board
                move_map[neighbor.board] = neighbor.path[-1]
                heapq.heappush(frontier, neighbor)

    return None

def reconstruct_path(parent_map, move_map, state):
    path_boards = []
    path_moves = []
    while parent_map[state] is not None:
        path_boards.append(state)
        path_moves.append(move_map[state])
        state = parent_map[state]
    path_boards.append(state)
    path_boards.reverse()
    path_moves.reverse()
    return path_boards, path_moves

def print_board(board):
    size = int(len(board) ** 0.5)
    for i in range(size):
        row = board[i*size:(i+1)*size]
        print(" ".join(str(x) if x != 0 else " " for x in row))
    print()
```

```
if __name__ == "__main__":
    initial_state = (1, 5, 8,
                     3, 2, 0,
                     4, 6, 7)

    final_state = (1, 2, 3,
                   4, 5, 6,
                   7, 8, 0)

    result = a_star(initial_state, final_state)
    if result:
        solution_boards, solution_moves = result
        print("Step-by-step solution:\n")
        for step_num, board in enumerate(solution_boards):
            moves_so_far = "".join(solution_moves[:step_num])
            print(f"Step {step_num}: Moves: {moves_so_far}")
            print_board(board)
            time.sleep(1)
    else:
        print("Name:Sujan G E ")
        print("USN:1BM23CS347")
        print("No solution found.")
```

Misplaced Tiles Heuristic (Hamming Distance)

```
import heapq
import time

class PuzzleState:
    def __init__(self, board, goal, path="", cost=0):
        self.board = board
        self.goal = goal
        self.path = path
        self.cost = cost
        self.zero_pos = self.board.index(0)
        self.size = int(len(board) ** 0.5)
```

```
    def __lt__(self, other):

        return (self.cost + self.heuristic()) < (other.cost + other.heuristic())

    def heuristic(self):

        misplaced = 0
        for i, tile in enumerate(self.board):
            if tile != 0 and tile != self.goal[i]:
                misplaced += 1
        return misplaced

    def get_neighbors(self):
        neighbors = []
        x, y = divmod(self.zero_pos, self.size)
        moves = {'U': (x - 1, y), 'D': (x + 1, y), 'L': (x, y - 1), 'R': (x, y + 1)}

        for move, (nx, ny) in moves.items():
            if 0 <= nx < self.size and 0 <= ny < self.size:
                new_zero_pos = nx * self.size + ny
                new_board = list(self.board)
                # Swap blank with the adjacent tile
                new_board[self.zero_pos], new_board[new_zero_pos] = new_board[new_zero_pos],
new_board[self.zero_pos]
                neighbors.append(PuzzleState(tuple(new_board), self.goal, self.path + move,
self.cost + 1))
        return neighbors

def a_star(start, goal):
    start_state = PuzzleState(start, goal)
    frontier = []
    heapq.heappush(frontier, start_state)
    explored = set()
    parent_map = {start_state.board: None}
    move_map = {start_state.board: ""}

    while frontier:
        current_state = heapq.heappop(frontier)

        if current_state.board == goal:
            return reconstruct_path(parent_map, move_map, current_state.board)

        explored.add(current_state.board)
```

```
        for neighbor in current_state.get_neighbors():
            if neighbor.board not in explored and neighbor.board not in parent_map:
                parent_map[neighbor.board] = current_state.board
                move_map[neighbor.board] = neighbor.path[-1]
                heapq.heappush(frontier, neighbor)

    return None

def reconstruct_path(parent_map, move_map, state):
    path_boards = []
    path_moves = []
    while parent_map[state] is not None:
        path_boards.append(state)
        path_moves.append(move_map[state])
        state = parent_map[state]
    path_boards.append(state)
    path_boards.reverse()
    path_moves.reverse()
    return path_boards, path_moves

def print_board(board):
    size = int(len(board) ** 0.5)
    for i in range(size):
        row = board[i*size:(i+1)*size]
        print(" ".join(str(x) if x != 0 else " " for x in row))
    print()

if __name__ == "__main__":
    initial_state = (2,8,3,
                     1, 6, 4,
                     7, 0,5)

    final_state = (1, 2, 3,
                   8, 0, 4,
                   7,6,5)

    result = a_star(initial_state, final_state)
    if result:
        solution_boards, solution_moves = result
        print("Step-by-step solution:\n")
        for step_num, board in enumerate(solution_boards):
            moves_so_far = "".join(solution_moves[:step_num])  # Moves taken to reach this step
            print(f"Step {step_num}: Moves: {moves_so_far}")
            print_board(board)
```

```
        time.sleep(1)  # Pause 1 second between steps for clarity
else:
    print("No solution found.")
```

```
Step-by-step solution:

Step 0: Moves:
2 8 3
1 6 4
7   5

Step 1: Moves: U
2 8 3
1   4
7 6 5

Step 2: Moves: UU
2   3
1 8 4
7 6 5

Step 3: Moves: UUL
  2 3
1 8 4
7 6 5

Step 4: Moves: UULD
1 2 3
  8 4
7 6 5

Step 5: Moves: UULDR
1 2 3
8   4
7 6 5
```

SUHAS BP (1BM23CS345)_