Faryal Rasuli, David Xing CSDC25BB

```
import heapq
import random
import time
import numpy as np
from memory_profiler import memory_usage


class PuzzleState:
    """
    Attributes:
        state (list): The current configuration of the 8-puzzle.
        goal_state (list): The target configuration that the puzzle is
trying to achieve.
        moves (int): The number of moves taken to reach this state from
the start state.
        prev (PuzzleState): Reference to the previous PuzzleState for
tracking the solution path.
        hamming (int): The hamming distance of the current state from
the goal state.
        manhattan (int): The manhattan distance of the current state
from the goal state.
    """

    def __init__(self, state, goal_state, moves=0, prev=None):
        self.state = state
        self.goal_state = goal_state
        self.moves = moves
        self.prev = prev
        self.hamming = self.calculate_hamming_distance()
        self.manhattan = self.calculate_manhattan_distance()

    def calculate_inversions(self):
        """
        Calculate the number of inversions in the current state. An
inversion is a pair of tiles
        that are in the reverse order from their order in the goal
state.

        Returns:
            int: The number of inversions.
        """
        inv_count = 0
        for i in range(len(self.state)):
            for j in range(i + 1, len(self.state)):
                if self.state[i] > self.state[j] != 0 and self.state[i]
```

```
!= 0:
                    inv_count += 1
        return inv_count

    def calculate_hamming_distance(self):
        """
        Calculate the Hamming distance of the current state from the
goal state. The Hamming distance
        is the number of tiles that are in the wrong position.

        Returns:
            int: The Hamming distance.
        """
        return sum(1 for i, tile in enumerate(self.state) if tile != 0
and tile != self.goal_state[i])

    def calculate_manhattan_distance(self):
        """
        Calculate the Manhattan distance of the current state from the
goal state. The Manhattan distance
        is the sum of the distances of each tile from its goal position.

        Returns:
            int: The Manhattan distance.
        """
        distance = 0
        for i, tile in enumerate(self.state):
            if tile != 0:
                goal_row, goal_col = divmod(self.goal_state.index(tile),
3)
                current_row, current_col = divmod(i, 3)
                distance += abs(goal_row - current_row) + abs(goal_col -
current_col)
        return distance

    def is_goal(self):
        """
        Check if the current state is the goal state.

        Returns:
            bool: True if the current state is the goal state, False
otherwise.
        """
        return self.state == self.goal_state

    def __lt__(self, other):
        """
        Compare two PuzzleStates based on their estimated cost to reach
the goal state.
        This is used by the priority queue to order states.

        Args:
            other (PuzzleState): The other PuzzleState to compare with.

        Returns:
            bool: True if the current state has a lower estimated cost
than the other state.
        """
        return (self.moves + self.manhattan) < (other.moves +
other.manhattan)


def get_neighbors(state):
    """
    Generate all possible moves (neighbors) from the current state by
sliding a tile into the blank space.

    Args:
        state (list): The current configuration of the 8-puzzle.

    Returns:
        list: A list of new states that can be reached from the current
state.
    """
    moves = []
    b_idx = state.index(0)  # Find the index of the blank tile
(represented by 0)
    b_row, b_col = divmod(b_idx, 3)  # Get the row and column of the
blank tile

    # Possible movements: up, down, left, right
    neighbors = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    for dr, dc in neighbors:
        new_row, new_col = b_row + dr, b_col + dc
        if 0 <= new_row < 3 and 0 <= new_col < 3:  # Ensure the move is
within bounds
            new_idx = new_row * 3 + new_col
            new_state = state[:]
            new_state[b_idx], new_state[new_idx] = new_state[new_idx],
new_state[
                b_idx]  # Swap the blank with the adjacent tile
            moves.append(new_state)
    return moves


def a_star(initial_state, goal_state, heuristic='manhattan'):
    """
    Solve the 8-puzzle game using the A* search algorithm.

    Args:
        initial_state (list): The starting configuration of the
8-puzzle.
        goal_state (list): The target configuration of the 8-puzzle.
        heuristic (str): The heuristic to use ('manhattan' or
'hamming').

    Returns:
        list: The sequence of moves from the initial state to the goal
state, or None if no solution.
    """
    start = PuzzleState(initial_state, goal_state)
    frontier = []  # Initialize the priority queue with the start state
    heapq.heappush(frontier, (0, start))
    explored = set()  # Set of explored states to avoid revisiting

    while frontier:
        cost, current_state = heapq.heappop(frontier)

        if current_state.is_goal():
            path = []
            while current_state.prev:
                path.append(current_state.state)
                current_state = current_state.prev
            return path[::-1]  # Return the path from the initial state
to the goal state

        explored.add(tuple(current_state.state))
        for neighbor in get_neighbors(current_state.state):
            if tuple(neighbor) not in explored:
                next_state = PuzzleState(neighbor, goal_state,
current_state.moves + 1, current_state)
                if heuristic == 'hamming':
                    heapq.heappush(frontier, (next_state.moves +
next_state.hamming, next_state))
                else:  # default is manhattan
                    heapq.heappush(frontier, (next_state.moves +
next_state.manhattan, next_state))
```

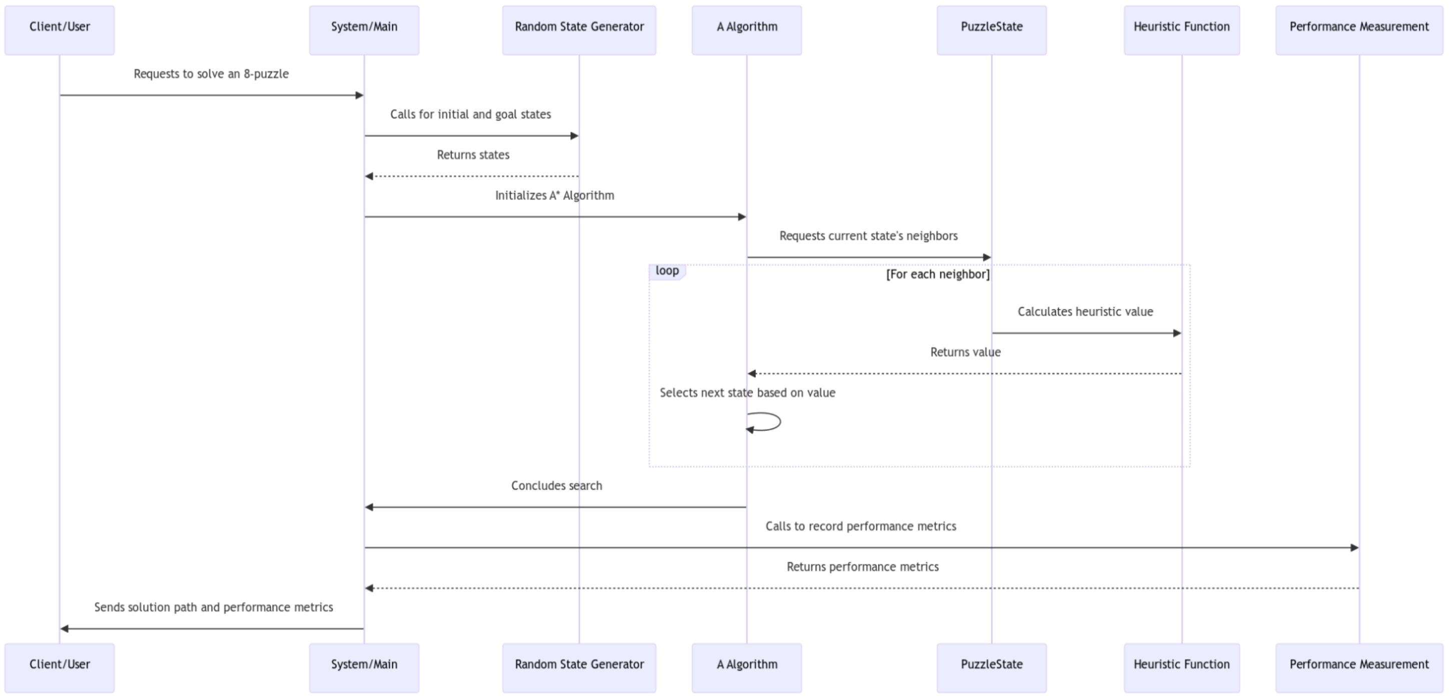
```
    return None  # Return None if no solution is found


def measure_performance(states, goal_state):
    """
    Measure the performance of the A* algorithm using both Hamming and
Manhattan heuristics.

    Args:
        states (list): A list of initial configurations to solve.
        goal_state (list): The target configuration for all the puzzles.

    Returns:
        dict: A dictionary containing the times and memory usage for
each heuristic.
    """
    results = {"hamming": {"times": [], "memory": []},
               "manhattan": {"times": [], "memory": []}}

    for i, state in enumerate(states):
        # Measure performance using Hamming heuristic
        start_time = time.time()
        mem_usage = memory_usage((a_star, (state, goal_state,
'hamming')), max_usage=True, retval=True)
        end_time = time.time()
        results["hamming"]["memory"].append(mem_usage[0])
        results["hamming"]["times"].append(end_time - start_time)

        # Measure performance using Manhattan heuristic
        start_time = time.time()
        mem_usage = memory_usage((a_star, (state, goal_state,
'manhattan')), max_usage=True, retval=True)
        end_time = time.time()
        results["manhattan"]["memory"].append(mem_usage[0])
        results["manhattan"]["times"].append(end_time - start_time)

        # After each solve, calculate and format the progress output
        progress_output = "{}/100".format(i + 1).rjust(7, ' ')  # Right
justify to match the length of "100/100"

        # Print a new line after every 10 puzzles for readability, also
ensure same length for each output
        if (i + 1) % 10 == 0:
            print(progress_output)  # New line after every 10th puzzle
        else:
            print(progress_output, end=' ', flush=True)  # Stay on the
same line for other puzzles

    return results


def generate_random_solvable_goal():
    """
    Generate a random solvable goal configuration for the 8-puzzle.

    Returns:
        list: A solvable configuration that can be used as a goal state.
    """
    canonical_goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]
    while True:
        state = random.sample(range(9), 9)
        if PuzzleState(state,
canonical_goal_state).calculate_inversions() % 2 == 0:
            return state


def generate_random_solvable_state_to_goal(goal_state):
    """
    Generate a random start state that is solvable to the specified goal
state.

    Args:
        goal_state (list): The goal configuration that the start state
needs to be solvable to.

    Returns:
        list: A random solvable start configuration.
    """
    while True:
        state = random.sample(range(9), 9)
        if PuzzleState(state, goal_state).calculate_inversions() % 2 ==
PuzzleState(goal_state,

goal_state).calculate_inversions() % 2:
            return state


if __name__ == '__main__':
    program_start_time = time.time()  # Capture the start time of the
program
```

```
    random_goal_state = generate_random_solvable_goal()  # Generate a
random solvable goal state

    random_states =
[generate_random_solvable_state_to_goal(random_goal_state) for _ in
                     range(100)]  # Generate 100 random solvable states

    performance_results = measure_performance(random_states,
                                                          random_goal_state)  #
Measure performance for both heuristics

    # Print statistics for Hamming heuristic
    print("\nHamming Statistics:")
    print("Time Mean:",
np.mean(performance_results["hamming"]["times"]))
    print("Time Std:", np.std(performance_results["hamming"]["times"]))
    print("Memory Mean:",
np.mean(performance_results["hamming"]["memory"]))
    print("Memory Std:",
np.std(performance_results["hamming"]["memory"]))

    # Print statistics for Manhattan heuristic
    print("\nManhattan Statistics:")
    print("Time Mean:",
np.mean(performance_results["manhattan"]["times"]))
    print("Time Std:",
np.std(performance_results["manhattan"]["times"]))
    print("Memory Mean:",
np.mean(performance_results["manhattan"]["memory"]))
    print("Memory Std:",
np.std(performance_results["manhattan"]["memory"]))

    program_end_time = time.time()  # Capture the end time of the
program
    runtime = program_end_time - program_start_time  # Calculate the
runtime
    print(f"\nProgram runtime: {runtime} seconds")  # Print the runtime
```

1. Short Task Description

a. The 8-puzzle problem involves sliding tiles on a 3x3 grid to achieve a specific goal state from a given random start state. This task implements the A* search algorithm with two different heuristics - Hamming and Manhattan - to solve the puzzle. The objective is to compare these heuristics in terms of memory usage (number of expanded nodes) and computation time across 100 random states.

2. Software Architecture Diagram


Client/User
System/Main
Random State Generator
A Algorithm
PuzzleState
Heuristic Function
Performance Measurement
Requests to solve an 8-puzzle
Calls for initial and goal states
Returns states
Initializes A* Algorithm
Requests current state's neighbors
loop
[For each neighbor]
Calculates heuristic value
Returns value
Selects next state based on value
Concludes search
Calls to record performance metrics
Returns performance metrics
Sends solution path and performance metrics

3. Short Descriptions of Modules and Interfaces

a. **PuzzleState Module:** Manages the state of the 8-puzzle, including the calculation of Hamming and Manhattan distances.
b. **Heuristic Functions:** Two functions to calculate Hamming (number of misplaced tiles) and Manhattan (total distance of tiles from their goal positions) distances.
c. **A* Algorithm Module:** Implements the A* search algorithm to find the shortest path to the goal state.
d. **Performance Measurement:** Measures and compares the performance of the two heuristics in terms of memory usage and computation time.
e.

4. Explain Design Decisions

a. Choice of Heuristics: Hamming and Manhattan distances were chosen due to their relevance and common usage in pathfinding algorithms.
b. Data Structures: Priority queues (heapq) for managing the frontier in A*, and sets for tracking explored states.

5. Discussion and Conclusions

a. Experience: The implementation highlighted the strengths and weaknesses of each heuristic. While Hamming appears faster on average (smaller mean time), it's also more variable (larger time standard deviation). Manhattan uses slightly less memory on average (smaller mean memory) and is slightly more consistent in memory usage (smaller memory standard deviation). Manhattan generally performed better in terms of both memory usage and time complexity.

b. Complexity Comparisons:

	Hamming	Manhattan
Time Mean Deviation	0.6576	0.7285
Time Standard Deviation	0.1420	0.0604
Memory Mean Deviation	66.2911	65.6663
Memory Standard Deviation	6.6473	6.5317

For Time: Smaller mean (average) time is generally better as it indicates faster solutions. Smaller standard deviation in time suggests more consistent solve times across different puzzles.

For Memory: Smaller mean memory usage is generally better as it indicates less resource consumption. Smaller standard deviation in memory suggests more consistent memory usage across different puzzles.

c. Possible Improvements: Findings of new Heuristics to improve computation time.