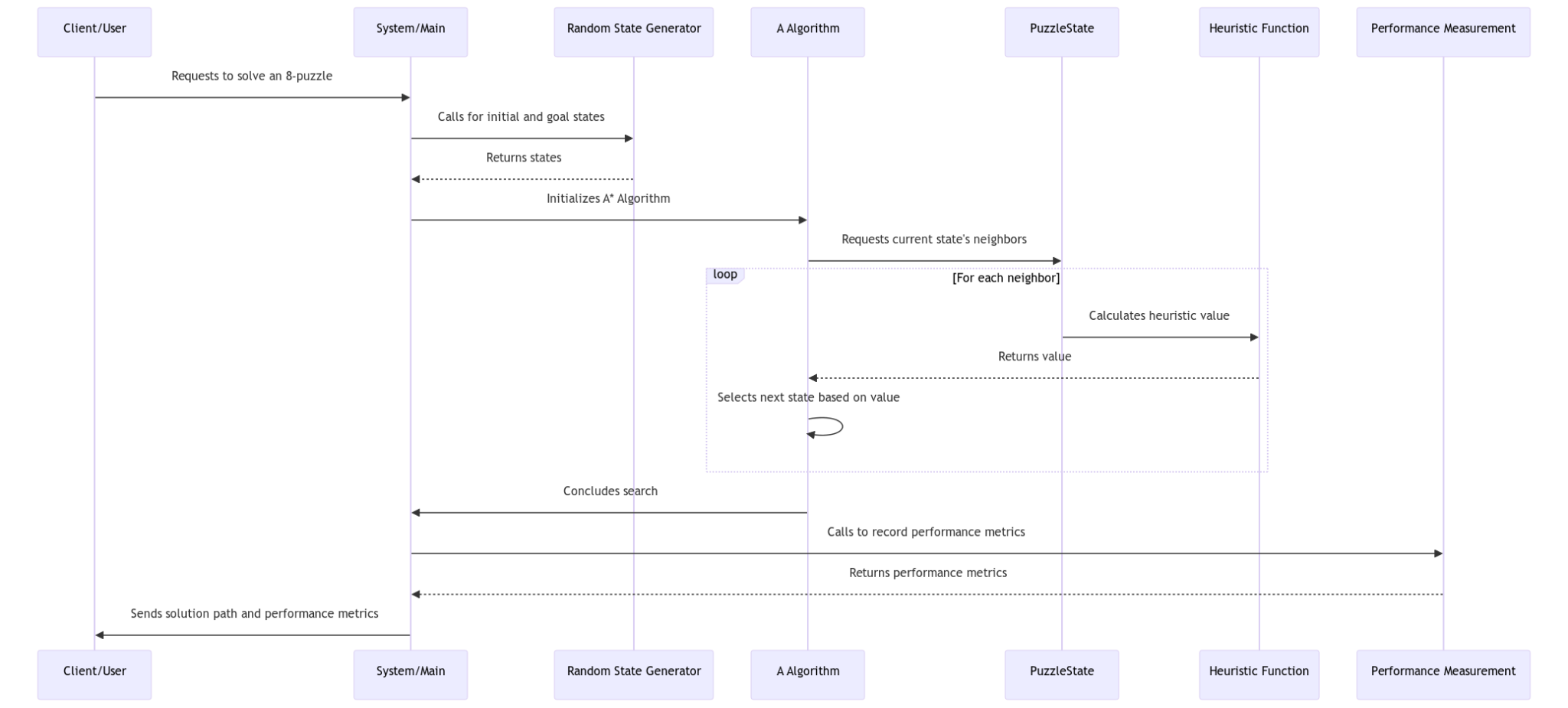
| import heapq import random import time import numpy as np from memory\_profiler import memory\_usage   class PuzzleState:  """  Attributes:  state (list): The current configuration of the 8-puzzle.  goal\_state (list): The target configuration that the puzzle is trying to achieve.  moves (int): The 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**
   1. 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**
3. **Short Descriptions of Modules and Interfaces**
   1. **PuzzleState Module:** Manages the state of the 8-puzzle, including the calculation of Hamming and Manhattan distances.
   2. **Heuristic Functions:** Two functions to calculate Hamming (number of misplaced tiles) and Manhattan (total distance of tiles from their goal positions) distances.
   3. **A\* Algorithm Module:** Implements the A\* search algorithm to find the shortest path to the goal state.
   4. **Performance Measurement:** Measures and compares the performance of the two heuristics in terms of memory usage and computation time.
4. **Explain Design Decisions**
   1. Choice of Heuristics: Hamming and Manhattan distances were chosen due to their relevance and common usage in pathfinding algorithms.
   2. Data Structures: Priority queues (heapq) for managing the frontier in A\*, and sets for tracking explored states.
5. **Discussion and Conclusions**
   1. 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.
   2. 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.

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