```
import heapq
import time
from collections import deque
from typing import List, Tuple, Optional, Set
import random

class PuzzleState:

    def __init__(self, board: List[int], size: int, g: int = 0, h: int =
                 parent: Optional['PuzzleState'] = None, move: str = "")
        self.board = board
        self.size = size
        self.g = g
        self.h = h
        self.f = g + h
        self.parent = parent
        self.move = move
        self.empty_pos = board.index(0)

    def __lt__(self, other):
        if self.f == other.f:
            return self.h < other.h
        return self.f < other.f

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

    def __hash__(self):
        return hash(tuple(self.board))

    def to_string(self) -> str:
        return ','.join(map(str, self.board))

def manhattan_distance(board: List[int], goal: List[int], size: int) ->
    distance = 0
    for i, tile in enumerate(board):
        if tile != 0:
            curr_row, curr_col = i // size, i % size
            goal_idx = goal.index(tile)
            goal_row, goal_col = goal_idx // size, goal_idx % size
            distance += abs(curr_row - goal_row) + abs(curr_col - goal_c
    return distance

def hamming_distance(board: List[int], goal: List[int]) -> int:
    return sum(1 for i, tile in enumerate(board) if tile != 0 and tile !
```

```
def generate_goal_board(size: int) -> List[int]:
    return list(range(1, size * size)) + [0]

def count_inversions(board: List[int]) -> int:
    inversions = 0
    board_without_zero = [x for x in board if x != 0]
    for i in range(len(board_without_zero)):
        for j in range(i + 1, len(board_without_zero)):
            if board_without_zero[i] > board_without_zero[j]:
                inversions += 1
    return inversions

def is_solvable(board: List[int], size: int) -> bool:
    inversions = count_inversions(board)

    if size % 2 == 1:
        return inversions % 2 == 0
    else:
        empty_row_from_bottom = size - (board.index(0) // size)
        if empty_row_from_bottom % 2 == 0:
            return inversions % 2 == 1
        else:
            return inversions % 2 == 0

def generate_random_board(size: int, max_attempts: int = 100) -> List[in
    for _ in range(max_attempts):
        board = list(range(size * size))
        random.shuffle(board)
        if is_solvable(board, size):
            return board

    goal = generate_goal_board(size)
    state = PuzzleState(goal[:], size)
    moves_count = size * size * 10

    for _ in range(moves_count):
        possible_moves = get_possible_moves(state)
        if possible_moves:
            move = random.choice(possible_moves)
            state = move

    return state.board

def get_possible_moves(state: PuzzleState) -> List[PuzzleState]:
    moves = []
    empty_pos = state.empty_pos
    row, col = empty_pos // state.size, empty_pos % state.size
    size = state.size
```

```
        directions = [
            ("UP", -1, 0),
            ("DOWN", 1, 0),
            ("LEFT", 0, -1),
            ("RIGHT", 0, 1)
        ]

        for direction, dr, dc in directions:
            new_row, new_col = row + dr, col + dc

            if 0 <= new_row < size and 0 <= new_col < size:
                new_pos = new_row * size + new_col
                new_board = state.board[:]
                new_board[empty_pos], new_board[new_pos] = new_board[new_pos

                new_state = PuzzleState(
                    board=new_board,
                    size=size,
                    g=state.g + 1,
                    parent=state,
                    move=direction
                )
                moves.append(new_state)

        return moves

    def solve_astar(initial_board: List[int], goal_board: List[int], size: i
                    heuristic: str = 'manhattan') -> Tuple[Optional[List[Puz
        start_time = time.time()

        if heuristic == 'manhattan':
            h_func = lambda board: manhattan_distance(board, goal_board, siz
        else:
            h_func = lambda board: hamming_distance(board, goal_board)

        initial_h = h_func(initial_board)
        initial_state = PuzzleState(initial_board, size, g=0, h=initial_h)

        open_list = [initial_state]
        closed_set: Set[str] = set()
        nodes_expanded = 0
        max_open_size = 1

        goal_str = ','.join(map(str, goal_board))

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

```
            nodes_expanded += 1

            if current.to_string() == goal_str:
                path = []
                node = current
                while node:
                    path.append(node)
                    node = node.parent
                path.reverse()

                end_time = time.time()
                stats = {
                    'algorithm': 'A*',
                    'heuristic': heuristic,
                    'nodes_expanded': nodes_expanded,
                    'solution_length': len(path) - 1,
                    'time_elapsed': end_time - start_time,
                    'max_open_size': max_open_size
                }
                return path, stats

            current_str = current.to_string()
            if current_str in closed_set:
                continue
            closed_set.add(current_str)

            for successor in get_possible_moves(current):
                successor_str = successor.to_string()

                if successor_str in closed_set:
                    continue

                successor.h = h_func(successor.board)
                successor.f = successor.g + successor.h

                heapq.heappush(open_list, successor)

            max_open_size = max(max_open_size, len(open_list))

        end_time = time.time()
        stats = {
            'algorithm': 'A*',
            'heuristic': heuristic,
            'nodes_expanded': nodes_expanded,
            'solution_length': None,
            'time_elapsed': end_time - start_time,
            'max_open_size': max_open_size
        }
        return None, stats
```

```
        return None, stats

    def solve_idastar(initial_board: List[int], goal_board: List[int], size:
                      heuristic: str = 'manhattan') -> Tuple[Optional[List[P
        start_time = time.time()

        if heuristic == 'manhattan':
            h_func = lambda board: manhattan_distance(board, goal_board, siz
        else:
            h_func = lambda board: hamming_distance(board, goal_board)

        goal_str = ','.join(map(str, goal_board))
        nodes_expanded = 0

        def search(path: List[PuzzleState], g: int, bound: int) -> Tuple[boo
            nonlocal nodes_expanded

            current = path[-1]
            nodes_expanded += 1

            f = g + current.h

            if f > bound:
                return False, f, nodes_expanded

            if current.to_string() == goal_str:
                return True, f, nodes_expanded

            min_bound = float('inf')

            for successor in get_possible_moves(current):
                if len(path) > 1 and successor.to_string() == path[-2].to_st
                    continue

                successor.h = h_func(successor.board)
                path.append(successor)

                found, new_bound, _ = search(path, g + 1, bound)

                if found:
                    return True, new_bound, nodes_expanded

                min_bound = min(min_bound, new_bound)
                path.pop()

            return False, min_bound, nodes_expanded

        initial_h = h_func(initial_board)
        initial_state = PuzzleState(initial_board, size, g=0, h=initial_h)
```

```
        bound = initial_h
        path = [initial_state]

        iterations = 0
        max_iterations = 100

        while iterations < max_iterations:
            iterations += 1
            found, new_bound, _ = search(path, 0, bound)

            if found:
                end_time = time.time()
                stats = {
                    'algorithm': 'IDA*',
                    'heuristic': heuristic,
                    'nodes_expanded': nodes_expanded,
                    'solution_length': len(path) - 1,
                    'time_elapsed': end_time - start_time,
                    'iterations': iterations
                }
                return path[:], stats

            if new_bound == float('inf'):
                break

            bound = new_bound
            path = [initial_state]

        end_time = time.time()
        stats = {
            'algorithm': 'IDA*',
            'heuristic': heuristic,
            'nodes_expanded': nodes_expanded,
            'solution_length': None,
            'time_elapsed': end_time - start_time,
            'iterations': iterations
        }
        return None, stats

    def print_board(board: List[int], size: int):
        for i in range(size):
            row = board[i*size:(i+1)*size]
            print("  " + " ".join(f"{tile:3}" if tile != 0 else "  ." for ti

    def print_solution(path: List[PuzzleState], size: int):
        if not path:
            print("No solution found!")
            return
```

```
    print(f"\n{'='*50}")
    print(f"SOLUTION PATH ({len(path)-1} moves)")
    print(f"{'='*50}\n")

    for i, state in enumerate(path):
        if i == 0:
            print(f"Step {i}: INITIAL STATE")
        else:
            print(f"Step {i}: Move {state.move}")

        print_board(state.board, size)
        print(f"  g={state.g}, h={state.h}, f={state.f}")
        print()

def print_statistics(stats: dict):
    print(f"\n{'='*50}")
    print(f"STATISTICS")
    print(f"{'='*50}")
    print(f"Algorithm:        {stats['algorithm']}")
    print(f"Heuristic:        {stats['heuristic']}")
    print(f"Nodes expanded:   {stats['nodes_expanded']}")

    if stats['solution_length'] is not None:
        print(f"Solution length:  {stats['solution_length']} moves")
    else:
        print(f"Solution:         NOT FOUND")

    print(f"Time elapsed:     {stats['time_elapsed']:.4f} seconds")

    if 'max_open_size' in stats:
        print(f"Max open list:    {stats['max_open_size']}")
    if 'iterations' in stats:
        print(f"IDA* iterations:  {stats['iterations']}")
    print(f"{'='*50}\n")

def solve_puzzle(size: int = 3, initial_board: Optional[List[int]] = Non
                 algorithm: str = 'astar', heuristic: str = 'manhattan',
                 show_solution: bool = True) -> Tuple[Optional[List[Puzz

    if initial_board is None:
        initial_board = generate_random_board(size)

    goal_board = generate_goal_board(size)

    print(f"\n{'='*50}")
    print(f"SLIDING PUZZLE SOLVER")
    print(f"{'='*50}")
    print(f"Board size:  {size}x{size}")
```

```
    print(f"Board size:  {size}x{size}")
    print(f"Algorithm:   {algorithm.upper()}")
    print(f"Heuristic:   {heuristic.upper()}")
    print(f"\nInitial state:")
    print_board(initial_board, size)
    print(f"\nGoal state:")
    print_board(goal_board, size)

    if not is_solvable(initial_board, size):
        print("\nWARNING: This configuration is NOT solvable!")
        return None, {}

    print("\nConfiguration is solvable. Starting search...\n")

    if algorithm == 'astar':
        path, stats = solve_astar(initial_board, goal_board, size, heuri
    else:
        path, stats = solve_idastar(initial_board, goal_board, size, heu

    print_statistics(stats)

    if show_solution and path:
        print_solution(path, size)
    elif path and not show_solution:
        print(f"Solution found! ({stats['solution_length']} moves)")
        print("Set show_solution=True to see the full path.")

    return path, stats

if __name__ == "__main__":
    print("="*70)
    print("SLIDING PUZZLE SOLVER - A* AND IDA* ALGORITHMS")
    print("Task 2 Implementation")
    print("="*70)

    print("\n\n### EXAMPLE 1: 3x3 Puzzle with A* (Manhattan) ###")
    solve_puzzle(size=3, algorithm='astar', heuristic='manhattan', show_

    print("\n\n### EXAMPLE 2: 3x3 Puzzle with IDA* (Hamming) ###")
    solve_puzzle(size=3, algorithm='idastar', heuristic='hamming', show_

    print("\n\n### EXAMPLE 3: 4x4 Puzzle with A* (Manhattan) ###")
    solve_puzzle(size=4, algorithm='astar', heuristic='manhattan', show_

    print("\n" + "="*70)
    print("To use this solver:")
    print("1. Call solve_puzzle() with desired parameters")
    print("2. Parameters:")
    print("   - size: 3, 4, or 5")
```

```
    print("   - size: 3, 4, or 5")
    print("   - algorithm: 'astar' or 'idastar'")
    print("   - heuristic: 'manhattan' or 'hamming'")
    print("   - initial_board: list of integers (None for random)")
    print("   - show_solution: True to see step-by-step solution")
    print("="*70)
```

```
======================================================================
SLIDING PUZZLE SOLVER - A* AND IDA* ALGORITHMS
Task 2 Implementation
======================================================================


### EXAMPLE 1: 3x3 Puzzle with A* (Manhattan) ###

==================================================
SLIDING PUZZLE SOLVER
==================================================
Board size:  3x3
Algorithm:   ASTAR
Heuristic:   MANHATTAN

Initial state:
    2   3   4
    1   7   5
    8   6   .

Goal state:
    1   2   3
    4   5   6
    7   8   .

Configuration is solvable. Starting search...


==================================================
STATISTICS
==================================================
Algorithm:        A*
Heuristic:        manhattan
Nodes expanded:   50
Solution length:  14 moves
Time elapsed:     0.0012 seconds
Max open list:    37
==================================================


==================================================
SOLUTION PATH (14 moves)
==================================================

Step 0: INITIAL STATE
    2   3   4
```

```
    1   7   5
    8   6   .
  g=0, h=12, f=12

Step 1: Move UP
    2   3   4
    1   7   .
    8   6   5
  g=1, h=13, f=14

Step 2: Move UP
    2   3   .
    1   7   4
```