```python
import heapq
import time
from collections import deque
import copy

class PuzzleState:
    def __init__(self, board, parent=None, move=""):
        self.board = board
        self.parent = parent
        self.move = move
        self.depth = 0
        if parent:
            self.depth = parent.depth + 1

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

    def __lt__(self, other):
        return False

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

    def get_blank_pos(self):
        for i in range(3):
            for j in range(3):
                if self.board[i][j] == 0:
                    return (i, j)

    def get_possible_moves(self):
        moves = []
        i, j = self.get_blank_pos()

        if i > 0:
            moves.append("UP")
        if i < 2:
            moves.append("DOWN")
        if j > 0:
            moves.append("LEFT")
        if j < 2:
            moves.append("RIGHT")

        return moves
```

```
    def generate_child(self, move):
        i, j = self.get_blank_pos()
        new_board = copy.deepcopy(self.board)

        if move == "UP":
            new_board[i][j], new_board[i-1][j] = new_board[i-1][j],
new_board[i][j]
        elif move == "DOWN":
            new_board[i][j], new_board[i+1][j] = new_board[i+1][j],
new_board[i][j]
        elif move == "LEFT":
            new_board[i][j], new_board[i][j-1] = new_board[i][j-1],
new_board[i][j]
        elif move == "RIGHT":
            new_board[i][j], new_board[i][j+1] = new_board[i][j+1],
new_board[i][j]

        return PuzzleState(new_board, self, move)

    def is_goal(self, goal):
        return self.board == goal

    def print_path(self):
        if self.parent:
            self.parent.print_path()
        print(f"Move: {self.move}")
        for row in self.board:
            print(row)
        print()

def misplaced_tiles(state, goal):
    count = 0
    for i in range(3):
        for j in range(3):
            if state.board[i][j] != goal[i][j] and state.board[i][j] !=
0:
                count += 1
    return count

def manhattan_distance(state, goal):
    distance = 0
    goal_pos = {}
```

```
    for i in range(3):
        for j in range(3):
            goal_pos[goal[i][j]] = (i, j)

    for i in range(3):
        for j in range(3):
            if state.board[i][j] != 0:
                goal_i, goal_j = goal_pos[state.board[i][j]]
                distance += abs(i - goal_i) + abs(j - goal_j)
    return distance

def bfs(initial_state, goal):
    start_time = time.time()
    visited = set()
    queue = deque([initial_state])
    nodes_expanded = 0

    while queue:
        current_state = queue.popleft()
        nodes_expanded += 1

        if current_state.is_goal(goal):
            end_time = time.time()
            return {
                "solution": current_state,
                "time": end_time - start_time,
                "nodes_expanded": nodes_expanded
            }

        visited.add(current_state)

        for move in current_state.get_possible_moves():
            child = current_state.generate_child(move)
            if child not in visited:
                queue.append(child)
                visited.add(child)

    return None

def greedy(initial_state, goal, heuristic):
    start_time = time.time()
    visited = set()
    priority_queue = []
```

```
    heapq.heappush(priority_queue, (heuristic(initial_state, goal),
initial_state))
    nodes_expanded = 0

    while priority_queue:
        _, current_state = heapq.heappop(priority_queue)
        nodes_expanded += 1

        if current_state.is_goal(goal):
            end_time = time.time()
            return {
                "solution": current_state,
                "time": end_time - start_time,
                "nodes_expanded": nodes_expanded
            }

        visited.add(current_state)

        for move in current_state.get_possible_moves():
            child = current_state.generate_child(move)
            if child not in visited:
                heapq.heappush(priority_queue, (heuristic(child, goal),
child))
                visited.add(child)

    return None

def astar(initial_state, goal, heuristic):
    start_time = time.time()
    visited = set()
    priority_queue = []
    heapq.heappush(priority_queue, (heuristic(initial_state, goal) +
initial_state.depth, initial_state))
    nodes_expanded = 0

    while priority_queue:
        _, current_state = heapq.heappop(priority_queue)
        nodes_expanded += 1

        if current_state.is_goal(goal):
            end_time = time.time()
            return {
                "solution": current_state,
```

```
                "time": end_time - start_time,
                "nodes_expanded": nodes_expanded
            }

        visited.add(current_state)

        for move in current_state.get_possible_moves():
            child = current_state.generate_child(move)
            if child not in visited:
                heapq.heappush(priority_queue, (heuristic(child, goal)
+ child.depth, child))
                visited.add(child)

    return None

def main():
    initial_board = [
        [1, 2, 3],
        [8, 0, 4],
        [7, 6, 5]
    ]
    goal_board = [
        [1, 2, 3],
        [8, 4, 0],
        [7, 6, 5]
    ]

    initial_state = PuzzleState(initial_board)

    print("Resolvendo com Busca em Largura...")
    bfs_result = bfs(initial_state, goal_board)
    if bfs_result:
        print(f"Tempo: {bfs_result['time']:.4f}s")
        print(f"Nós expandidos: {bfs_result['nodes_expanded']}")
        print("Solução encontrada:")
        bfs_result['solution'].print_path()
    else:
        print("Solução não encontrada")

    print("\nResolvendo com Busca Gulosa (Peças fora do lugar)...")
    greedy_result = greedy(initial_state, goal_board, misplaced_tiles)
    if greedy_result:
        print(f"Tempo: {greedy_result['time']:.4f}s")
```

```
        print(f"Nós expandidos: {greedy_result['nodes_expanded']}")
        print("Solução encontrada:")
        greedy_result['solution'].print_path()
    else:
        print("Solução não encontrada")

    print("\nResolvendo com A* (Distância de Manhattan)...")
    astar_manhattan_result = astar(initial_state, goal_board,
manhattan_distance)
    if astar_manhattan_result:
        print(f"Tempo: {astar_manhattan_result['time']:.4f}s")
        print(f"Nós expandidos:
{astar_manhattan_result['nodes_expanded']}")
        print("Solução encontrada:")
        astar_manhattan_result['solution'].print_path()
    else:
        print("Solução não encontrada")

    print("\nResolvendo com A* (Peças fora do lugar)...")
    astar_misplaced_result = astar(initial_state, goal_board,
misplaced_tiles)
    if astar_misplaced_result:
        print(f"Tempo: {astar_misplaced_result['time']:.4f}s")
        print(f"Nós expandidos:
{astar_misplaced_result['nodes_expanded']}")
        print("Solução encontrada:")
        astar_misplaced_result['solution'].print_path()
    else:
        print("Solução não encontada")

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