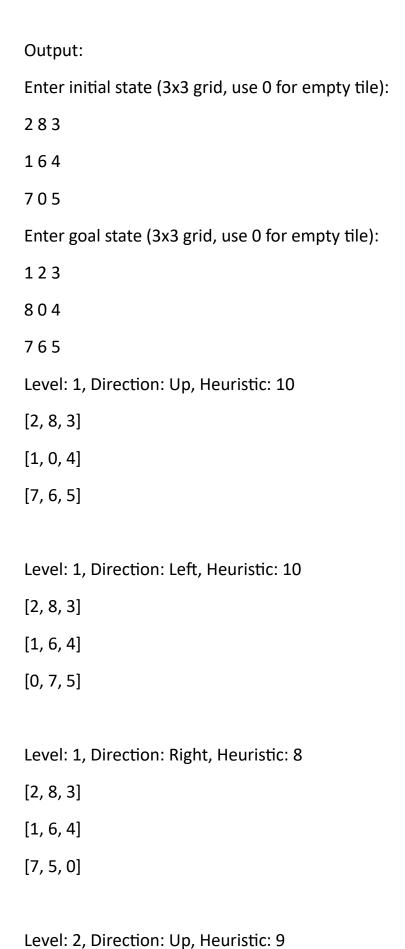
```
Python code for 8-puzzle A* implementation, to calculate, f(n),
considering:
g(n): Depth of the node, h(n): Manhattan Distance
import heapq
class PuzzleState:
  def __init__(self, state, empty_tile_pos, g, h, path, level):
    self.state = state
    self.empty_tile_pos = empty_tile_pos # (row, col)
    self.g = g # Cost from start to current state
    self.h = h # Heuristic cost to goal
    self.f = g + h \# Total cost
    self.path = path # Path taken to reach this state
    self.level = level # Depth level in the state space
  def It (self, other):
    return self.f < other.f # Priority queue comparison based on f value
def astar_manhattan_distance(start_state, goal_state):
  directions = {
    (-1, 0): 'Up',
    (1, 0): 'Down',
    (0, -1): 'Left',
    (0, 1): 'Right'
  }
```

```
def calculate heuristic(state):
    h = 0
    for i in range(3):
       for j in range(3):
         value = state[i][j]
         if value != 0:
           goal_row = (value - 1) // 3
           goal_col = (value - 1) % 3
           h += abs(goal_row - i) + abs(goal_col - j)
    return h
  def generate_moves(state, empty_tile_pos):
    moves = []
    row, col = empty tile pos
    for (dr, dc), direction in directions.items():
       new_row, new_col = row + dr, col + dc
       if 0 \le \text{new row} \le 3 and 0 \le \text{new col} \le 3:
         new_state = [list(r) for r in state] # Deep copy
         # Swap the empty tile with the adjacent tile
         new_state[row][col], new_state[new_row][new_col] =
new_state[new_row][new_col], new_state[row][col]
         moves.append((new state, (new row, new col), direction))
    return moves
```

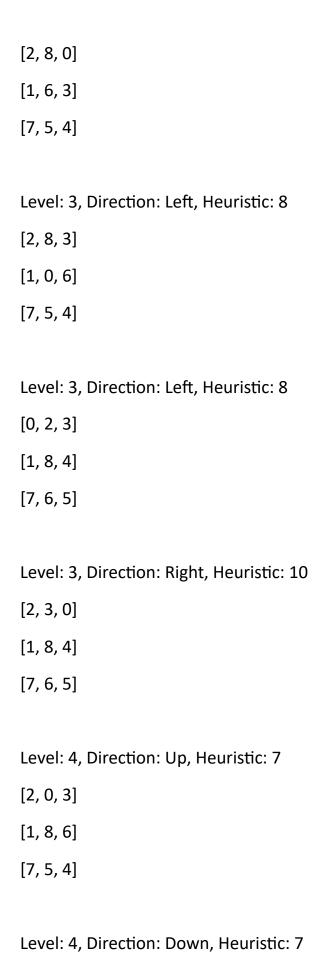
```
visited = set()
  start_empty_pos = next((i, j) for i in range(3) for j in range(3) if
start_state[i][j] == 0)
  start_h = calculate_heuristic(start_state)
  start_node = PuzzleState(start_state, start_empty_pos, 0, start_h,
[start state], 0)
  priority_queue = []
  heapq.heappush(priority_queue, start_node)
  while priority_queue:
    current_node = heapq.heappop(priority_queue)
    # Check if we reached the goal
    if current_node.state == goal_state:
      print("Goal state reached!")
      for step in current node.path:
         for row in step:
           print(row)
         print()
      return
    visited.add(tuple(map(tuple, current_node.state))) # Add current state to
visited
    # Generate possible moves
```

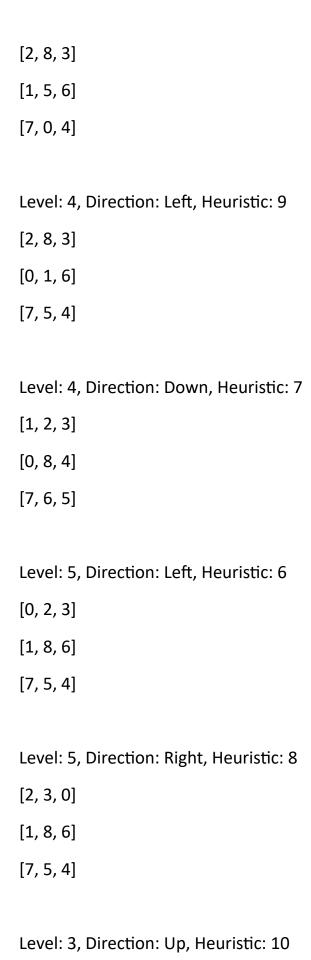
```
for new state, new empty pos, direction in
generate_moves(current_node.state, current_node.empty_tile_pos):
      if tuple(map(tuple, new_state)) not in visited:
        g = current_node.g + 1 # Cost from start
        h = calculate heuristic(new state) # Heuristic
        new_path = current_node.path + [new_state]
         new_node = PuzzleState(new_state, new_empty_pos, g, h, new_path,
current node.level + 1)
        # Print state information
         print(f"Level: {new_node.level}, Direction: {direction}, Heuristic:
{new_node.h}")
        for row in new node.state:
           print(row)
         print()
        heapq.heappush(priority queue, new node)
def main():
  print("Enter initial state (3x3 grid, use 0 for empty tile):")
  start_state = [list(map(int, input().split())) for _ in range(3)]
  print("Enter goal state (3x3 grid, use 0 for empty tile):")
  goal state = [list(map(int, input().split())) for in range(3)]
  astar manhattan distance(start state, goal state)
if name == " main ":
  main()
```

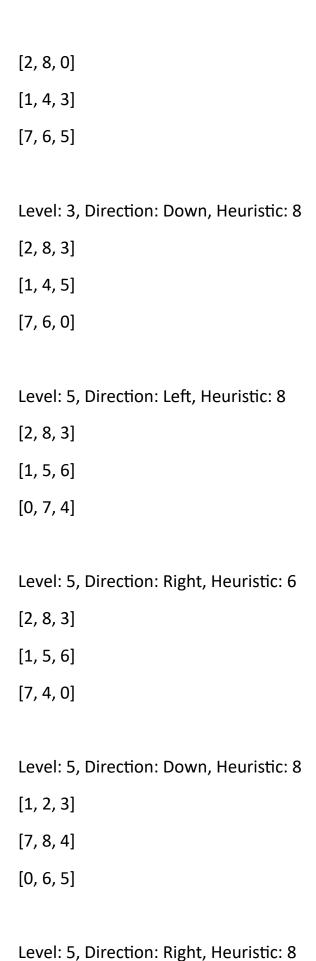


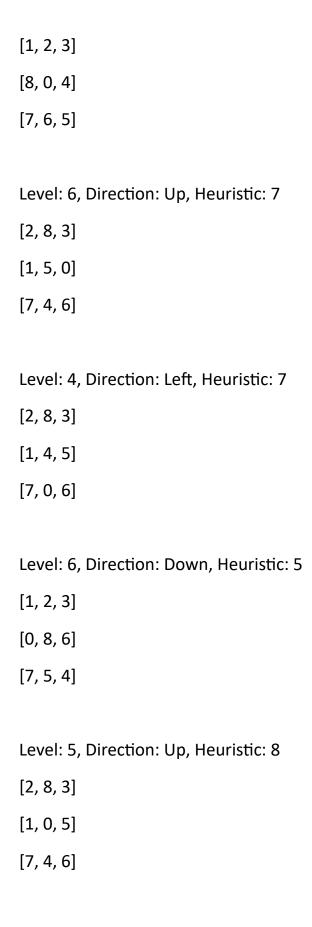
[2, 8, 3]	
[1, 6, 0]	
[7, 5, 4]	
Level: 2, Direction: Up, Heuristic: 11	L
[2, 8, 3]	
[0, 6, 4]	
[1, 7, 5]	
Level: 2, Direction: Up, Heuristic: 9	
[2, 0, 3]	
[1, 8, 4]	
[7, 6, 5]	
Level: 2, Direction: Left, Heuristic: 1	. 1
[2, 8, 3]	
[0, 1, 4]	
[7, 6, 5]	
Level: 2, Direction: Right, Heuristic:	ç
[2, 8, 3]	
[1, 4, 0]	
[7, 6, 5]	

Level: 3, Direction: Up, Heuristic: 10





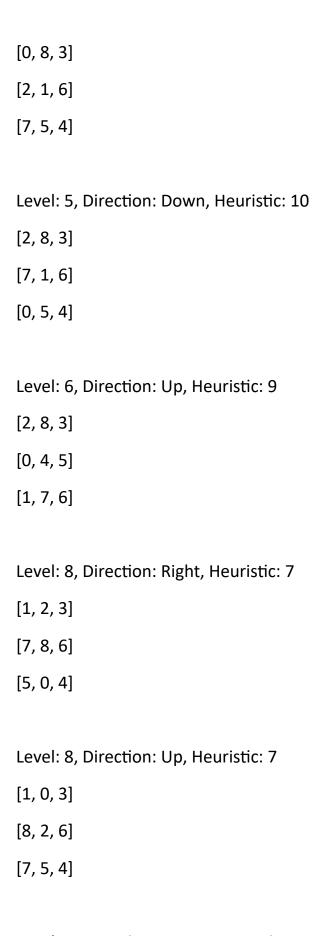




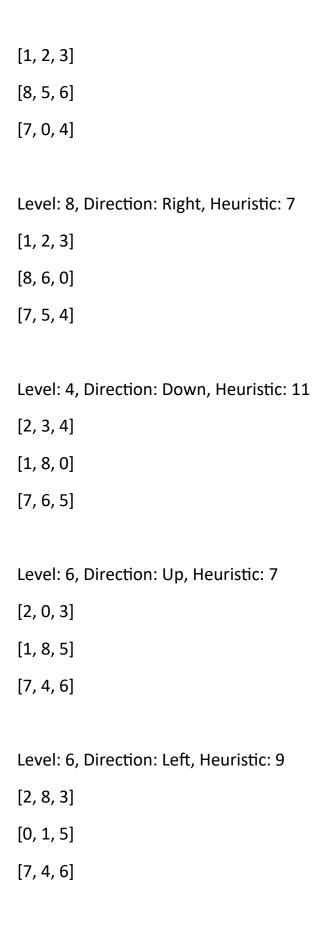
Level: 5, Direction: Left, Heuristic: 8

[2, 8, 3]
[1, 4, 5]
[0, 7, 6]
Level: 7, Direction: Down, Heuristic: 6
[1, 2, 3]
[7, 8, 6]
[0, 5, 4]
Level: 7, Direction: Right, Heuristic: 6
[1, 2, 3]
[8, 0, 6]
[7, 5, 4]
Level: 3, Direction: Up, Heuristic: 12
[0, 8, 3]
[2, 6, 4]
[1, 7, 5]
Level: 3, Direction: Right, Heuristic: 12
[2, 8, 3]
[6, 0, 4]
[1, 7, 5]

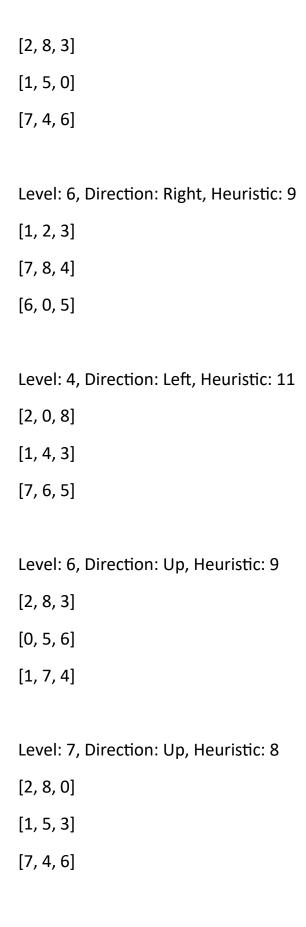
Level: 5, Direction: Up, Heuristic: 10



Level: 8, Direction: Down, Heuristic: 5



Level: 6, Direction: Right, Heuristic: 7



Level: 7, Direction: Up, Heuristic: 8

[2, 8, 0]
[1, 5, 3]
[7, 4, 6]
Level: 4, Direction: Left, Heuristic: 11
[2, 0, 8]
[1, 6, 3]
[7, 5, 4]
Level: 6, Direction: Down, Heuristic: 9
[2, 3, 6]
[1, 8, 0]
[7, 5, 4]
Goal state reached!
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]

- [2, 0, 3]
- [1, 8, 4]
- [7, 6, 5]
- [0, 2, 3]
- [1, 8, 4]
- [7, 6, 5]
- [1, 2, 3]
- [0, 8, 4]
- [7, 6, 5]
- [1, 2, 3]
- [8, 0, 4]
- [7, 6, 5]