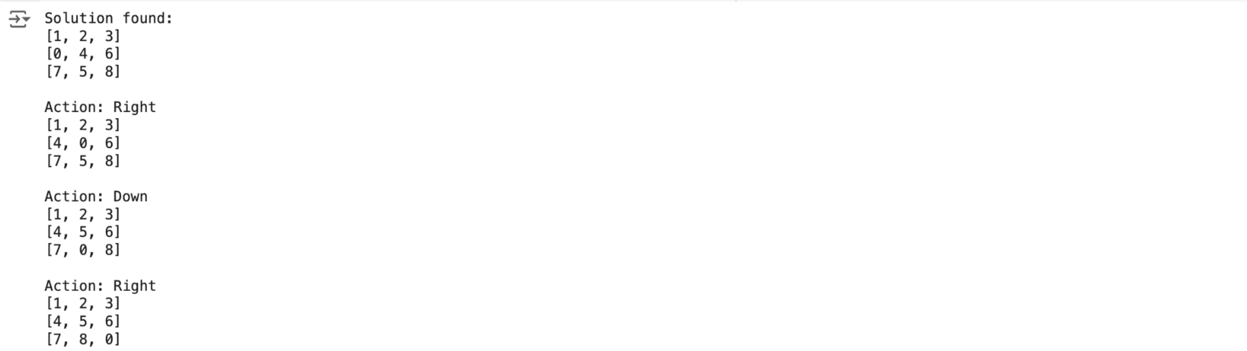
Implementation of Iterative deepening search algorithm.

Code:

```
import copy

class Node:
    def __init__(self, state, parent=None, action=None, depth=0):
        self.state = state
        self.parent = parent
        self.action = action
        self.depth = depth

    def __lt__(self, other):
        return self.depth < other.depth

    def expand(self):
        children = []
        row, col = self.find_blank()
        possible_actions = []

        if row > 0:  # Can move the blank tile up
            possible_actions.append('Up')
        if row < 2:  # Can move the blank tile down
            possible_actions.append('Down')
        if col > 0:  # Can move the blank tile left
            possible_actions.append('Left')
        if col < 2:  # Can move the blank tile right
            possible_actions.append('Right')

        for action in possible_actions:
            new_state = copy.deepcopy(self.state)
            if action == 'Up':
                new_state[row][col], new_state[row - 1][col] =
new_state[row - 1][col], new_state[row][col]
            elif action == 'Down':
                new_state[row][col], new_state[row + 1][col] =
new_state[row + 1][col], new_state[row][col]
            elif action == 'Left':
                new_state[row][col], new_state[row][col - 1] =
new_state[row][col - 1], new_state[row][col]
```

```
            elif action == 'Right':
                new_state[row][col], new_state[row][col + 1] =
new_state[row][col + 1], new_state[row][col]

            children.append(Node(new_state, self, action, self.depth + 1))
        return children

    def find_blank(self):
        for row in range(3):
            for col in range(3):
                if self.state[row][col] == 0:
                    return row, col

def depth_limited_search(node, goal_state, limit):
    if node.state == goal_state:
        return node
    if node.depth >= limit:
        return None
    for child in node.expand():
        result = depth_limited_search(child, goal_state, limit)
        if result is not None:
            return result
    return None

def iterative_deepening_search(initial_state, goal_state, max_depth):
    for depth in range(max_depth):
        result = depth_limited_search(Node(initial_state), goal_state,
depth)
        if result is not None:
            return result
    return None

def print_solution(node):
    path = []
    while node is not None:
        path.append((node.action, node.state))
        node = node.parent
    path.reverse()

    for action, state in path:
        if action:
            print(f"Action: {action}")
        for row in state:
            print(row)
        print()

initial_state = [[1, 2, 3], [0, 4, 6], [7, 5, 8]]
goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

max_depth = 20
solution = iterative_deepening_search(initial_state, goal_state,
max_depth)

if solution:
    print("Solution found:")
    print_solution(solution)
else:
    print("Solution not found.")
```

OUTPUT:



```
Solution found:
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]

Action: Right
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

Action: Down
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

Action: Right
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
```

⇒ ITERATIVE DEEPNING SEARCH.

1. Node class:

   • Represents the puzzle state, including the current configuration, parent node, action taken, and depth in search tree.

2. Methods:

   expand():

   generates child nodes by moving the blank tile in valid directions

   find-blank():

   locates the blank tile(0) in the grid.

3. Depth- limited search:

   • Recursively explores nodes up to a specific depth limit.

   • If the goal state is found return the node

   • If the depth limit is reached, return node

   • otherwise, expand child nodes and continue searching.

4. Iterative Deepening Search:

   • Iteratively increases the depth limit.

   • For each depth 0 to max-depth, perform ~~depth~~ - limited search.

   • If a solution is found return it

15/10/24

5. Backtrack from the goal node to print the sequence of actions and resulting states.