Program Title: vacuum cleaner agent

Code :

```
def vacuum_cleaner_agent(percept):
  """
  A simple vacuum cleaner agent that operates in a two-location world.

  Args:
    percept: A list containing the current location and whether it is dirty.
         e.g., ['A', 'Dirty']

  Returns:
    The action to be taken by the agent (Left, Right, Suck, NoOp).
  """

  location, status = percept

  if status == 'Dirty':
    return 'Suck'
  elif location == 'A':
    return 'Right'
  elif location == 'B':
    return 'Left'
  else:
    return 'NoOp'  # Should not reach here in this simple world.


# Example percept sequence and action execution
percepts = [['A', 'Clean'], ['A', 'Dirty'], ['B', 'Clean'], ['B', 'Dirty'], ['A', 'Clean'], ['A', 'Clean']]
actions = []

for percept in percepts:
```

```python
  action = vacuum_cleaner_agent(percept)

  actions.append(action)

  print(f"Percept: {percept}, Action: {action}")


print("\nPercept Sequence:", percepts)

print("Action Sequence:", actions)
```

Output:

```
Percept: ['A', 'Clean'], Action: Right
Percept: ['A', 'Dirty'], Action: Suck
Percept: ['B', 'Clean'], Action: Left
Percept: ['B', 'Dirty'], Action: Suck
Percept: ['A', 'Clean'], Action: Right
Percept: ['A', 'Clean'], Action: Right

Percept Sequence: [['A', 'Clean'], ['A', 'Dirty'], ['B', 'Clean'], ['B', 'Dirty'], ['A', 'Clean'], ['A', 'Clean']]
Action Sequence: ['Right', 'Suck', 'Left', 'Suck', 'Right', 'Right']
```

Algorithm:

Program Title: vacuum cleaner agent

Algorithm:

1. Sense the environment:
   - The agent preceives two things:
     - Its current location (A or B)
     - The status of that location (Dirty or clean)

2) Decision process:
   - If the current location is Dirty:
     - The agent takes the action suck to clean the location
   - Else if the current location is A.
     - The agent moves Right to location B.
   - Else if the current location is B:
     - The agent moves left to location A.

3) Act:
   - The agent performs the action based on its perception and the decision rules.

4) Repeat:
   - After the action is taken, the agent repeats the process when a new percept is received.

Output:

percept : ['A', 'clean'], Action: Right
percept : ['A', 'Dirty'], Action: suck
percept : ['B', 'clean'], Action: left
percept : ['B', 'Dirty'], Action: suck
percept : ['A', 'clean'], Action: Right
percept : ['A', 'clean'], Action: Right.

percept sequence: [['A', 'clean'], ['A', 'Dirty'], ['B', 'clean'], ['B', 'Dirty'], ['A', 'clean'], ['A', 'clean']]

Action sequence: ['Right', 'suck', 'left', 'suck', 'Right', 'Right'

LAB 2:

Program title: Solve 8 puzzle problems, Implement Iterative deepening search algorithm.

code:

```
import copy


# Directions for movement: up, down, left, right

moves = {'up': (-1, 0), 'down': (1, 0), 'left': (0, -1), 'right': (0, 1)}


# Check if a state is the goal state

def is_goal(state, goal_state):

    return state == goal_state


# Get the position of the empty space (0)

def get_empty_position(state):

    for i in range(3):

        for j in range(3):

            if state[i][j] == 0:

                return i, j


# Move the empty space in a specified direction if possible

def move_tile(state, direction):

    new_state = copy.deepcopy(state)

    empty_i, empty_j = get_empty_position(state)

    di, dj = moves[direction]

    new_i, new_j = empty_i + di, empty_j + dj

    if 0 <= new_i < 3 and 0 <= new_j < 3:

        new_state[empty_i][empty_j], new_state[new_i][new_j] = new_state[new_i][new_j],
new_state[empty_i][empty_j]
```

```
        return new_state
    return None


# Depth-limited search
def depth_limited_search(state, goal_state, depth_limit, path):
    if is_goal(state, goal_state):
        return state, path


    if depth_limit == 0:
        return None, []


    empty_i, empty_j = get_empty_position(state)
    for direction in moves:
        new_state = move_tile(state, direction)
        if new_state is not None and new_state not in path:  # Avoid loops
            result, new_path = depth_limited_search(new_state, goal_state, depth_limit - 1, path +
[new_state])
            if result:
                return result, new_path


    return None, []


# Iterative deepening search
def iterative_deepening_search(initial_state, goal_state):
    depth = 0
    while True:
        result, path = depth_limited_search(initial_state, goal_state, depth, [initial_state])
        if result is not None:
            return path, depth
        depth += 1
```

```
# Print the state of the puzzle
def print_state(state):
    for row in state:
        print(row)
    print()

# Test the 8-puzzle
initial_state = [
    [1, 2, 3],
    [4, 0, 5],
    [6, 7, 8]
]

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

# Solve the puzzle using iterative deepening search
solution_path, depth = iterative_deepening_search(initial_state, goal_state)

# Output the steps
print(f"Solution found in {depth} steps.\n")
print("Steps to reach the goal:")

for i, state in enumerate(solution_path):
    print(f"Step {i}:")
    print_state(state)
```

Output:

```
Solution found in 14 steps.

Steps to reach the goal:
Step 0:
[1, 2, 3]
[4, 0, 5]
[6, 7, 8]

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

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

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

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

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

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

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

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

Step 9:
[1, 2, 3]
[5, 6, 0]
[4, 7, 8]

Step 10:
[1, 2, 3]
[5, 0, 6]
[4, 7, 8]
```

```
Step 11:
[1, 2, 3]
[0, 5, 6]
[4, 7, 8]

Step 12:
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]

Step 13:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

Step 14:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
```

Algorithm:

8/10/24

program Title: Implement Iterative deepening search algorithm:

1. Initialization:

. Represent the puzzle as a 3x3 grid with 0 as the empty space.

• Define the goal state where titles are ordered from 1to8 with 0 in the bottom-right corner.

• Define valid moves for the empty space: up, down, left, right.

2) check Goal:

• compare the current puzzle state with the goal state

3) Get empty position:

• Locate the position of the empty space (0)

4) Move the empty space:

. Try moving the empty space in all directions and generate new valid state.

5) Depth-limited search (DLS):

• Explore states up to a given depth limit.

• If the goal isn't reach within the limit, backtrack and try new paths.

6) Iterative Deepening search (IDS):

• start with depth 0 and incrementally increase the depth limit.

• perform DLS for each limit until the goal state is found.

7. print solution:

• once the goal is found, print the sequence of steps leading from the initial state to the goal state.

output:

solution found in 14 steps

steps to reach the goal:

step 0:

$$\begin{bmatrix} 1, & 2, & 3 \\ 4, & 0, & 5 \\ 6, & 7, & 8 \end{bmatrix}$$

step 1:

$$\begin{bmatrix} 1, & 2, & 3 \\ 4, & 5, & 0 \\ 6, & 7, & 8 \end{bmatrix}$$

step 2:

$$\begin{bmatrix} 1, & 2, & 3 \\ 4, & 5, & 8 \\ 6, & 7, & 0 \end{bmatrix}$$

step 3:

$$\begin{bmatrix} 1, & 2, & 3 \\ 4, & 5, & 8 \\ 6, & 0, & 7 \end{bmatrix}$$

step 4:

$$\begin{bmatrix} 1, & 2, & 3 \\ 4, & 5, & 8 \\ 0, & 6, & 7 \end{bmatrix}$$

step 5:

$$\begin{bmatrix} 1, & 2, & 3 \\ 0, & 5, & 8 \\ 4, & 6, & 7 \end{bmatrix}$$

step 6:

$$\begin{bmatrix} 1, & 2, & 3 \\ 5, & 0, & 8 \\ 4, & 6, & 7 \end{bmatrix}$$

step 7:

$$\begin{bmatrix} 1, & 2, & 3 \\ 5, & 6, & 8 \\ 4, & 0, & 7 \end{bmatrix}$$

step 8:

$$\begin{bmatrix} 1, & 2, & 3 \\ 5, & 6, & 8 \\ 4, & 7, & 0 \end{bmatrix}$$

step 9:

$$\begin{bmatrix} 1, & 2, & 3 \\ 5, & 6, & 0 \\ 4, & 7, & 8 \end{bmatrix}$$

step 10:

$$\begin{bmatrix} 1 & 2 & 3 \\ 5 & 0 & 6 \\ 4 & 7 & 8 \end{bmatrix}$$

step 11:

$$\begin{bmatrix} 1, & 2, & 3 \\ 0, & 5, & 6 \\ 4, & 7, & 8 \end{bmatrix}$$

step 12:

$$\begin{bmatrix} 1, & 2, & 3 \\ 4, & 5, & 6 \\ 0, & 7, & 8 \end{bmatrix}$$

step 13:

$$\begin{bmatrix} 1, & 2, & 3 \\ 4, & 5, & 6 \\ 7, & 0, & 8 \end{bmatrix}$$

step 14:

$$\begin{bmatrix} 1, & 2, & 3 \\ 4, & 5, & 6 \\ 7, & 8, & 0 \end{bmatrix}$$

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]
```

```python
            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:
```

```python
        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]
```

Week 3:

A*_MisplaceTiles

CODE:

```
#Heauristic approach to 8-puzzle problem

import heapq

def solve_8puzzle(initial_state):
    goal_state = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]
    priority_queue = [(heuristic(initial_state, goal_state), 0, initial_state, [])]
    visited = set()

    while priority_queue:
        f_cost, g_cost, current_state, current_path = heapq.heappop(priority_queue)

        if current_state == goal_state:
            return current_path + [current_state]

        if tuple(map(tuple, current_state)) in visited:
            continue
        visited.add(tuple(map(tuple, current_state)))

        for next_state, action in get_possible_moves(current_state):
            new_g_cost = g_cost + 1
            new_f_cost = new_g_cost + heuristic(next_state, goal_state)
            heapq.heappush(priority_queue, (new_f_cost, new_g_cost, next_state,
current_path + [(current_state, action)]))

    return None


def heuristic(state, goal_state):
    misplaced_tiles = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != goal_state[i][j] and state[i][j] != 0:
                misplaced_tiles += 1
```

```
    return misplaced_tiles


def find_position(state, tile):
    for i in range(3):
        for j in range(3):
            if state[i][j] == tile:
                return i, j


def get_possible_moves(state):
    row, col = find_position(state, 0)
    possible_moves = []

    if row > 0:
        new_state = [list(row) for row in state]
        new_state[row][col], new_state[row - 1][col] = new_state[row - 1][col],
new_state[row][col]
        possible_moves.append((new_state, 'Up'))
    if row < 2:
        new_state = [list(row) for row in state]
        new_state[row][col], new_state[row + 1][col] = new_state[row + 1][col],
new_state[row][col]
        possible_moves.append((new_state, 'Down'))
    if col > 0:
        new_state = [list(row) for row in state]
        new_state[row][col], new_state[row][col - 1] = new_state[row][col - 1],
new_state[row][col]
        possible_moves.append((new_state, 'Left'))
    if col < 2:
        new_state = [list(row) for row in state]
        new_state[row][col], new_state[row][col + 1] = new_state[row][col + 1],
new_state[row][col]
        possible_moves.append((new_state, 'Right'))

    return possible_moves


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

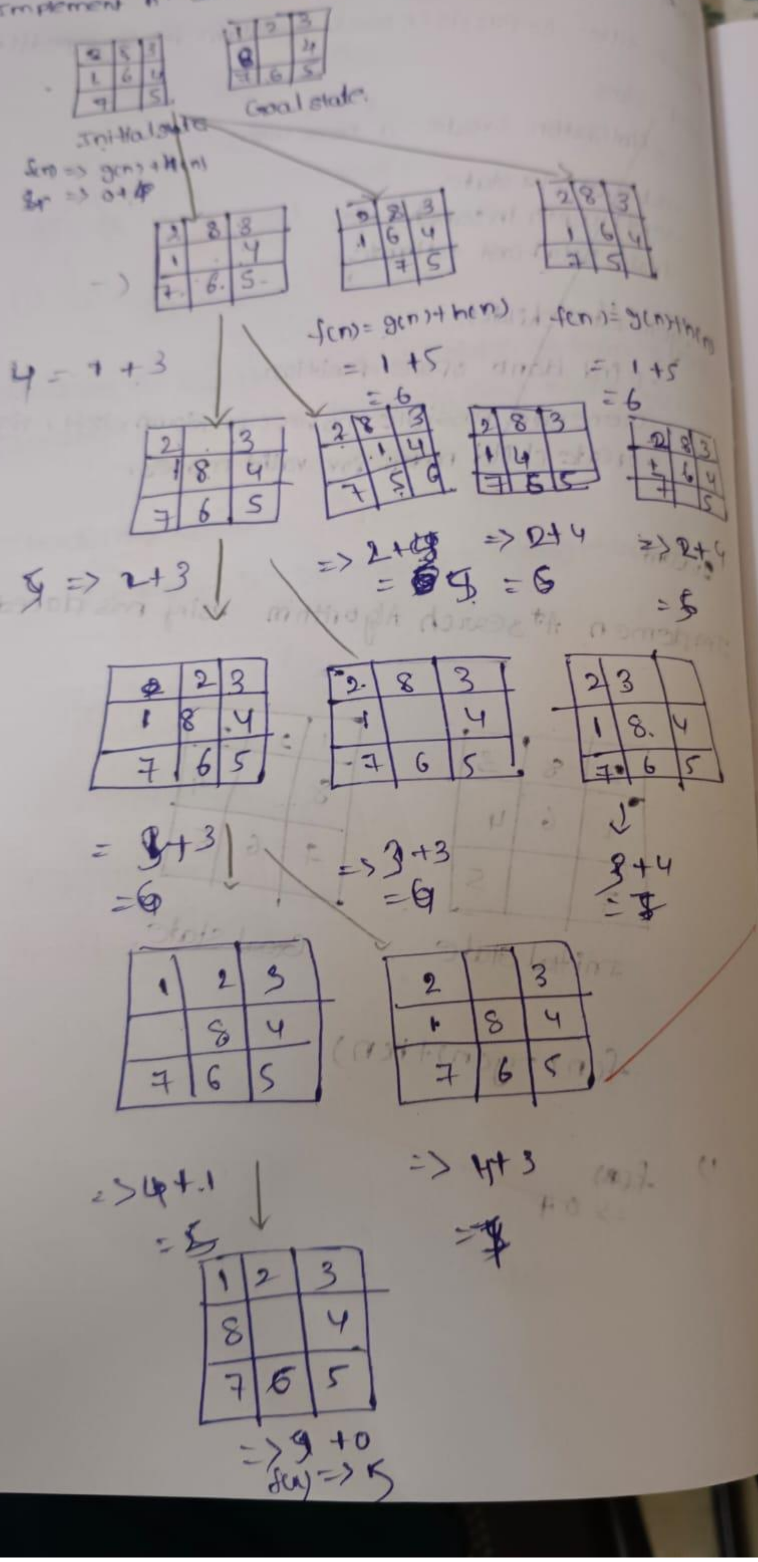
```
solution = solve_8puzzle(initial_state)

if solution:
    print("Solution found:")
    for state, action in solution[:-1]:
        print("--------------------")
        for row in state:
            print(row)
        print("Move:", action)
    print("--------------------")
    for row in solution[-1]:
        print(row)
else:
    print("No solution found.")
```

Output:

```
Solution found:
--------------------
[2, 8, 3]
[1, 6, 4]
[0, 7, 5]
Move: Right
--------------------
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]
Move: Up
--------------------
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]
Move: Up
--------------------
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]
Move: Left
--------------------
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]
Move: Down
--------------------
[1, 2, 3]
[0, 8, 4]
[7, 6, 5]
Move: Right
--------------------
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]
```

Implement A* search Algorithm using misplaced tiles



Algorithm:

Initialize

- start with th...
- set the goal ...

in the top-le...

2. Priority Queue

- use a priorit...

the puzzle prior...

- g(n) ...

start

- h(n) ...

counts how ...

3. Explore st...

- Rem...

the que...

if the...

the solu...

- otherwi...

moving ...

4. Evalu...

Algorithm:

Initialize:

- Start with the initial state of the puzzle.
- Set the goal state (which is when the black tile is in the top-left corner and all tiles are in order).

2. Priority Queue:

- use a priority queue (or min-heap) to store states of the puzzle, prioritized by $f(n) = g(n) + h(n)$:
  - $g(n)$ is the number of move (steps) taken from start.
  - $h(n)$ is the misplaced tiles heuristic, which counts how many tiles are not in their goal position.

3. Explore states:

- Remove the state with smallest $f(n)$ from the queue.
- if this state is the goal state, stop and return the solution.
- otherwise, generate all possible new states by moving the blank tile up, down, left or right.

4. Evaluate New state's:

- For each new state
  - calculate $g(n)$ (number of steps taken so far).
  - calculate $h(n)$ (number of misplaced tiles in the new state)
  - Add this new state to the priority queue with its $f(n) = g(n) + h(n)$

5. Repeat.

- Continue exploring states from the queue until the goal state is reached.

6. Goal Reached.

- Once the goal state is reached, the algorithm terminates and outputs the solution.

Implement A* search Algorithm using manhattan d[istance]

2	8	3
1	6	4
7		5

Initial state

1	2	3
8		4
7	6	5

Goal state.

(= g(n) + h(n))

= 0 + 5

= 5

2	8	3
1		4
7	6	5

=> 1 + 4

= 5

2	8	3
1	6	4
	7	5

2+1+1+1+1+1

=> 1 + 6

= 7

2	8	3
1	6	4
7	5	

1+1+1+2+1+1

=> ~~1 + 6~~

= 7

2		3
1	8	4
7	6	5

1+1+1+1

=> 2+3

= 5

2	8	3
	1	4
7	5	6

1+1+2+2+1

=> 2 + 5

= 7

2	8	3
1	4	
7	6	5

1+1+1+1+2

= 2 + 5

= 7

2	8	3
1	6	4
7		5

1+1+2+1

= 2+5

= 7

Week 3:

A*_ManhattanDistanceA

CODE:

```
#Manhattan approach
import heapq


def solve_8puzzle(initial_state):
    goal_state = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]
    priority_queue = [(heuristic(initial_state, goal_state), 0, initial_state, [])]
    visited = set()

    while priority_queue:
        f_cost, g_cost, current_state, current_path = heapq.heappop(priority_queue)

        if current_state == goal_state:
            return current_path + [current_state]

        if tuple(map(tuple, current_state)) in visited:
            continue
        visited.add(tuple(map(tuple, current_state)))

        for next_state, action in get_possible_moves(current_state):
            new_g_cost = g_cost + 1
            new_f_cost = new_g_cost + heuristic(next_state, goal_state)
            heapq.heappush(priority_queue, (new_f_cost, new_g_cost, next_state,
current_path + [(current_state, action)]))

    return None


def heuristic(state, goal_state):
    distance = 0
    for i in range(3):
```

```
        for j in range(3):
            if state[i][j] != 0:
                goal_row, goal_col = find_position(goal_state, state[i][j])
                distance += abs(i - goal_row) + abs(j - goal_col)
    return distance


def find_position(state, tile):
    for i in range(3):
        for j in range(3):
            if state[i][j] == tile:
                return i, j


def get_possible_moves(state):
    row, col = find_position(state, 0)
    possible_moves = []

    if row > 0:
        new_state = [list(row) for row in state]
        new_state[row][col], new_state[row - 1][col] = new_state[row - 1][col],
new_state[row][col]
        possible_moves.append((new_state, 'Up'))
    if row < 2:
        new_state = [list(row) for row in state]
        new_state[row][col], new_state[row + 1][col] = new_state[row + 1][col],
new_state[row][col]
        possible_moves.append((new_state, 'Down'))
    if col > 0:
        new_state = [list(row) for row in state]
        new_state[row][col], new_state[row][col - 1] = new_state[row][col - 1],
new_state[row][col]
        possible_moves.append((new_state, 'Left'))
    if col < 2:
        new_state = [list(row) for row in state]
```

```
        new_state[row][col], new_state[row][col + 1] = new_state[row][col + 1],
new_state[row][col]
        possible_moves.append((new_state, 'Right'))

    return possible_moves


initial_state = [[2, 8, 3], [1, 6, 4], [0, 7, 5]]
solution = solve_8puzzle(initial_state)

if solution:
    print("Solution found:")
    for state, action in solution[:-1]:
        print("--------------------")
        for row in state:
            print(row)
        print("Move:", action)
    print("--------------------")
    for row in solution[-1]:
        print(row)
else:
    print("No solution found.")
```

Output:

```
Solution found:
--------------------
[2, 8, 3]
[1, 6, 4]
[0, 7, 5]
Move: Right
--------------------
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]
Move: Up
--------------------
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]
Move: Up
--------------------
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]
Move: Left
--------------------
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]
Move: Down
--------------------
[1, 2, 3]
[0, 8, 4]
[7, 6, 5]
Move: Right
--------------------
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]
```

5. Repeat.

• Continue exploring states from the queue until the goal state is reached.

6. Goal Reached.

• Once the goal state is reached, the algorithm terminates and outputs the solution.

Implement A* search Algorithm using manhattan

2	8	3
1	6	4
7		5

Initial state

1	2	3
8		4
7	6	5

Goal state.

(= g(n)+h(n))
=0+5
=5

2	8	3
1		4
7	6	5

=> 1+4
=5

2	8	3
1	6	4
	7	5

2+1+1+1+1+1

=> 1+6
=7

2	8	3
1	6	4
7	5	

1+1+1+2+1+1

=> ~~1+6~~
=7

2		3
1	8	4
7	6	5

1+1+1+1
=> 2+3
=5

2	8	3
	1	4
7	5	6

1+1+2+1
=> 2+5
=7

2	8	3
1	4	
7	6	5

1+1+1+1+2
=2+5
=7

2	8	3
1	6	4
7		5

1+1+2+1
=
2+5
=7

	2	3
1	8	4
7	6	5

1+1+1

=> 3+2

= 5

2	8	3
1		4
7	6	5

1+1+1+2

=> 3+4

= 7

2	3	
1	8	4
7	6	5

1+1+1+1

=> 3+4

= 7

1	2	3
	8	4
7	6	5

1

= 4+1

= 5

2		3
1	8	4
7	6	5

1+1+1

= 4+3

= 7

1	2	3
8		4
7	6	5

=> = 5+0

= 5

Algorithm:

1. Initialize:
   - start with the Initial state of the puzzle
   - set the goal state

2. Priority queue:
   - use a priority queue to store states of the puzzle, prioritized by $f(n) = g(n) + h(n)$:
     - $g(n)$: The number of moves taken from the start
     - $h(n)$: The Manhattan distance heuristic, which sums the distances of tile from its correct position.

3. Explore states:

· Remove the state with the smallest f(n) from the queue.

· If the current state is the goal state, stop and return the solution.

· Generate all possible new state by moving the blank tile up, down, left or right.

4. Evaluate New states.

· For each new state:

· calculate g(n) (the depth)

· calculate h(n) the Manhattan distance

· Add this new state to the priority queue with its f(n) = g(n) + h(n)

5. Repeat:

· continue exploring states from the queue until the goal state is reached.

6. Goal Reached:

· once the goal state is reached, the algorithm terminates and outputs the solution.

Rm 15/10/24