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:

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