**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 = []

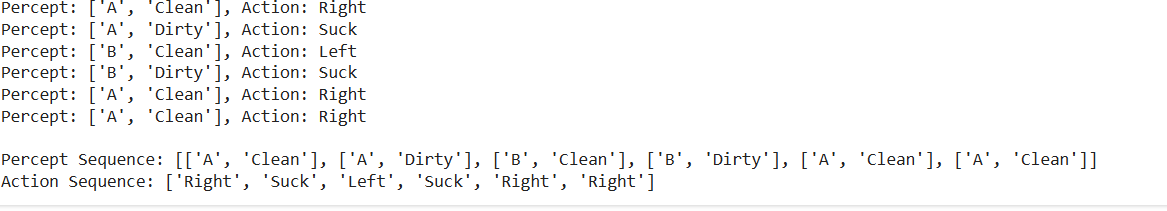
for percept in percepts:

action = vacuum\_cleaner\_agent(percept) actions.append(action)

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

print("\nPercept Sequence:", percepts) print("Action Sequence:", actions)

**Output:**



# Algorithm:

**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)

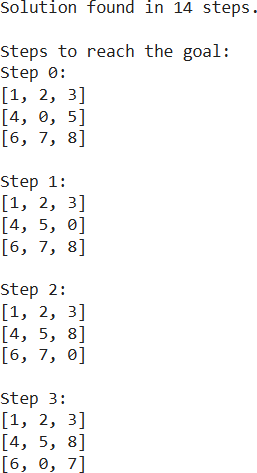
# Output the steps

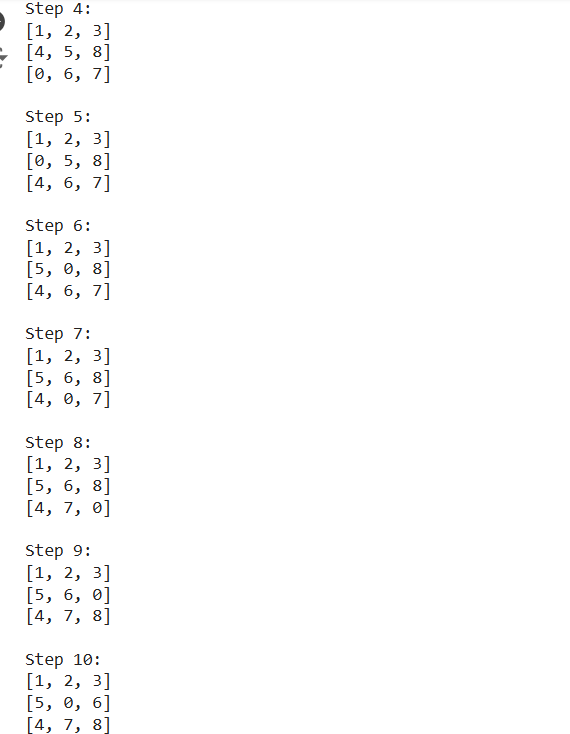
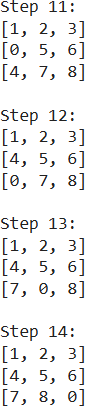
print(f"Solution found in {depth} steps.\n") print("Steps to reach the goal:")

for i, state in enumerate(solution\_path): print(f"Step {i}:")

print\_state(state)

**Output:**



**Algorithm:**

