LABORATORY PROGRAM – 1

IMPLEMENT TIC –TAC –TOE GAME

PSEUDOCODE OR ALGORITHM

01/10/24

Bafna Gold
Date: Page:

LAB-1

* Implement Tic-Tac-Toe Game.

Algorithm:

Step-1: Create a 3x3 board initialized with empty strings.

Step-2: Start with player 'x'

Step-3: Print the current board & then prompt the current player to enter their move (row and column).

Step-4: Check if the chosen cell is empty. If empty, place player's symbol in that cell.

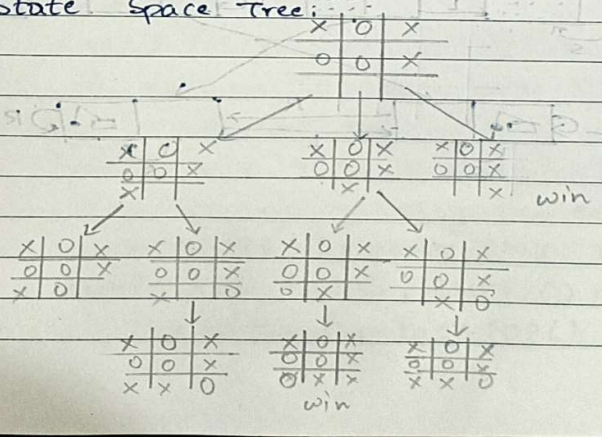
Step-5: Now, check for winner. If there is a winner print the board and declare the winner.

If the board is full and there's no winner, declare a tie.

Else, if the game is still going on, switch to the other player.

If the cell is not empty, prompt for a valid move.

State Space Tree:



CODE

```
      def print_board(board):
  for row in board:
    print("|".join(row))
    print("-"*5)
def check_winner(board):
  for i in range(3):
    if board[i][0]==board[i][1]==board[i][2]!=" ":
      return board[i][0]
    if board[0][i]==board[1][i]==board[2][i]!=" ":
      return board[0][i]

  if board[0][0]==board[1][1]==board[2][2] !=" ":
    return board[0][0]
  if board[0][2]==board[1][1]==board[2][0] !=" ":
    return board[0][2]
  return None
  def is_full(board):
    return all(cell !="" for row in board for cell in row )

  def tic_tac_toe():
    board=[]
    for _ in range(3):
      row=[]
      for _ in range(3):
        row.append("")
      board.append(row)
    current_player="X"
    while True:
        print_board(board)
        row = int(input(f"Player {current_player}, Enter the row (0-2): "))
        col = int(input(f"Player {current_player}, Enter the column (0-2): "))

        if board[row][col] == " ":
            board[row][col] = current_player
            winner = check_winner(board)
            if winner:
                print_board(board)
                print(f"Player {winner} wins!")
                break
            elif is_full(board):
                print_board(board)
                print("It's a tie!")
                break
            current_player = "O" if current_player == "X" else "X"
        else:
            print("Invalid move, try again.")
tic_tac_toe()
```

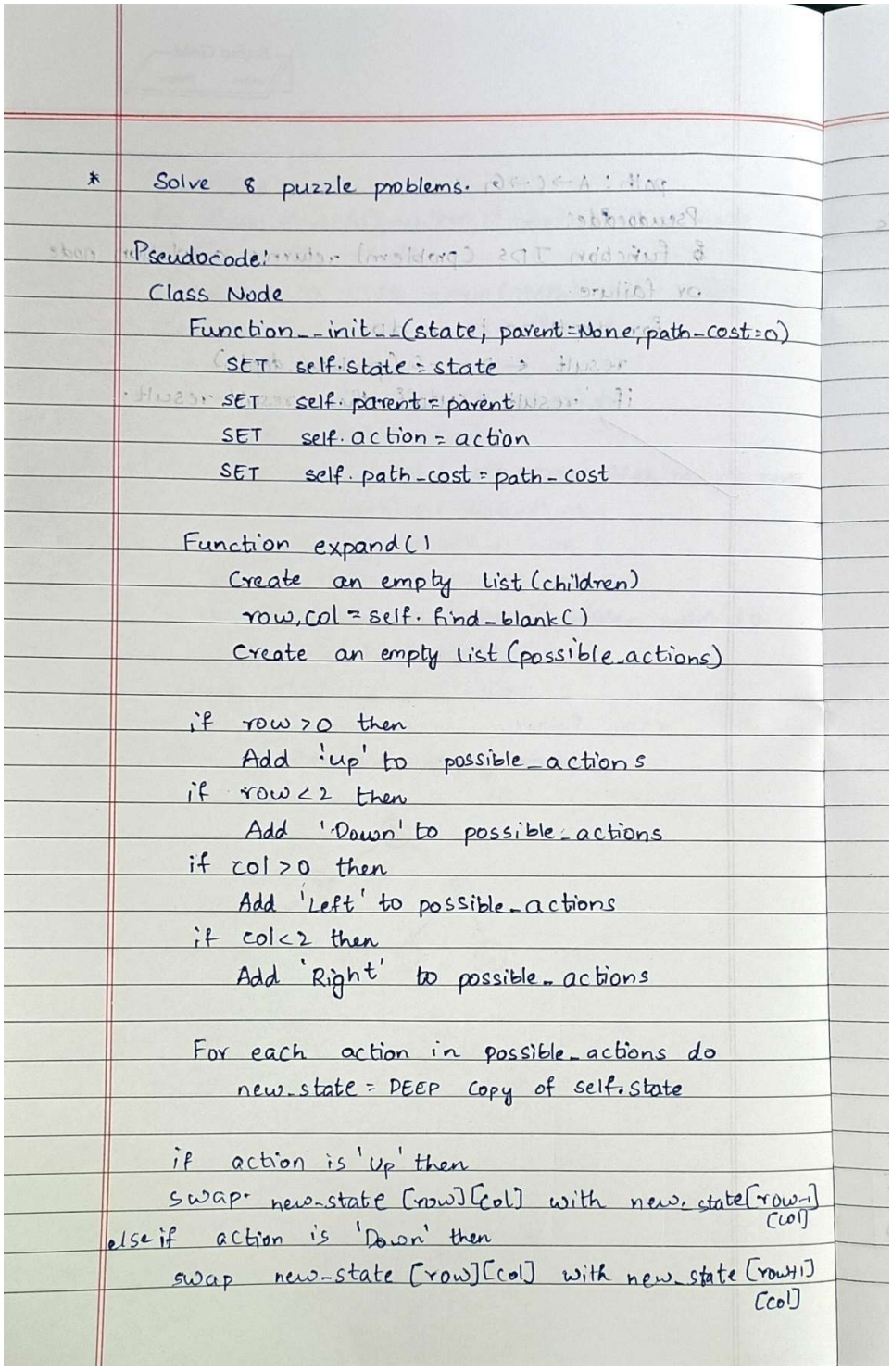
OUTPUT

```
 | |
-----
 | |
-----
 | |
-----
Player X, enter the row (0-2): 0
Player X, enter the column (0-2): 0
X| |
-----
 | |
-----
 | |
-----
Player O, enter the row (0-2): 1
Player O, enter the column (0-2): 1
X| |
-----
 |O|
-----
 | |
-----
Player X, enter the row (0-2): 0
Player X, enter the column (0-2): 2
X| |X
-----
 |O|
-----
 | |
-----
Player O, enter the row (0-2): 1
Player O, enter the column (0-2): 2
X| |X
-----
 |O|O
-----
 | |
-----
Player X, enter the row (0-2): 0
Player X, enter the column (0-2): 1
X|X|X
-----
 |O|O
-----
 | |
-----
Player X wins!
```

LABORATORY PROGRAM – 2

SOLVE 8 PUZZLE PROBLEMS

PSEUDOCODE OR ALGORITHM



* Solve 8 puzzle problems.

Pseudocode:

```
Class Node
    Function __init__(state, parent=None, path_cost=0)
        SET self.state = state
        SET self.parent = parent
        SET self.action = action
        SET self.path_cost = path_cost

    Function expand()
        Create an empty list (children)
        row, col = self.find_blank()
        Create an empty list (possible_actions)

    if row > 0 then
        Add 'up' to possible_actions
    if row < 2 then
        Add 'Down' to possible_actions
    if col > 0 then
        Add 'Left' to possible_actions
    if col < 2 then
        Add 'Right' to possible_actions

    For each action in possible_actions do
        new_state = DEEP copy of self.state

    if action is 'Up' then
    swap new_state [row][col] with new_state[row-1][col]
else if action is 'Down' then
    swap new_state [row][col] with new_state [row+1][col]
```

```
elseif action is 'Left' then
    Swap new_state[row][col] with new_state[row
                                            [col
else if action is 'Right' then
    swap new_state[row][col] with new_state[row]
                                            [col+
    Append new Node(new_state, self, action, self.path
        +1) to children
    End For
    Return children
End Function

Function find_blank()
    For row from 0 to 2 do
        For col from 0 to 2 do
            if self.state[row][col] == 0 then
                Return row,col
        End for
    End function
End Class

Function breadth_first_search(initial_state,
                              goal_state)
    Create a queue (frontier) and Enqueue
                the initial node.
    Create an empty set (explored)
    While frontier is not empty do
        node = Dequeue from frontier
    if node.state is equal to goal_state then
        Return node
    Add tuple representation of node.state to
                explored.
```

```
    For each child in node.expand() do
        if tuple representation of child.state is
        not in explored then
            Enqueue child into frontier
    End while
    Return None
End Function

Function print_solution(node)
    Create an empty list (path)
    While node is not None do
        Append (node.action, node.state) to path
        node = node.parent
    Reverse path
    For each (action, state) in path do
        if action is not None then
            Print "Action:" + action
        For each row in state do
            print row
        Print empty line
End Function
```

State Space Tree:

Initial state

1	2	3
4	5	6
7	8	0

1	2	3
4	5	6
7	0	8

1	2	3
4	0	6
7	5	8

1	2	3
4	5	6
0	7	8

1	2	3
4	5	6
7	8	0

CODE

```
import copy
from collections import deque
class Node:
    def __init__(self, state, parent=None, action=None, path_cost=0):
        self.state = state
        self.parent = parent
        self.action = action
        self.path_cost = path_cost
    def expand(self):
        children = []
        row, col = self.find_blank()
        possible_actions = []
        if row > 0:
            possible_actions.append('Up')
        if row < 2:
            possible_actions.append('Down')
        if col > 0:
            possible_actions.append('Left')
        if col < 2:
            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.path_cost + 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 breadth_first_search(initial_state, goal_state):
    frontier = deque([Node(initial_state)])
    explored = set()

    while frontier:
        node = frontier.popleft()
        if node.state == goal_state:
            return node
```

```
        explored.add(tuple(map(tuple, node.state)))
        for child in node.expand():
            if tuple(map(tuple, child.state)) not in explored:
                frontier.append(child)
    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]]

solution = breadth_first_search(initial_state, goal_state)
if solution:
    print("Solution found:")
    print_solution(solution)
else:
    print("Solution not found.")
```

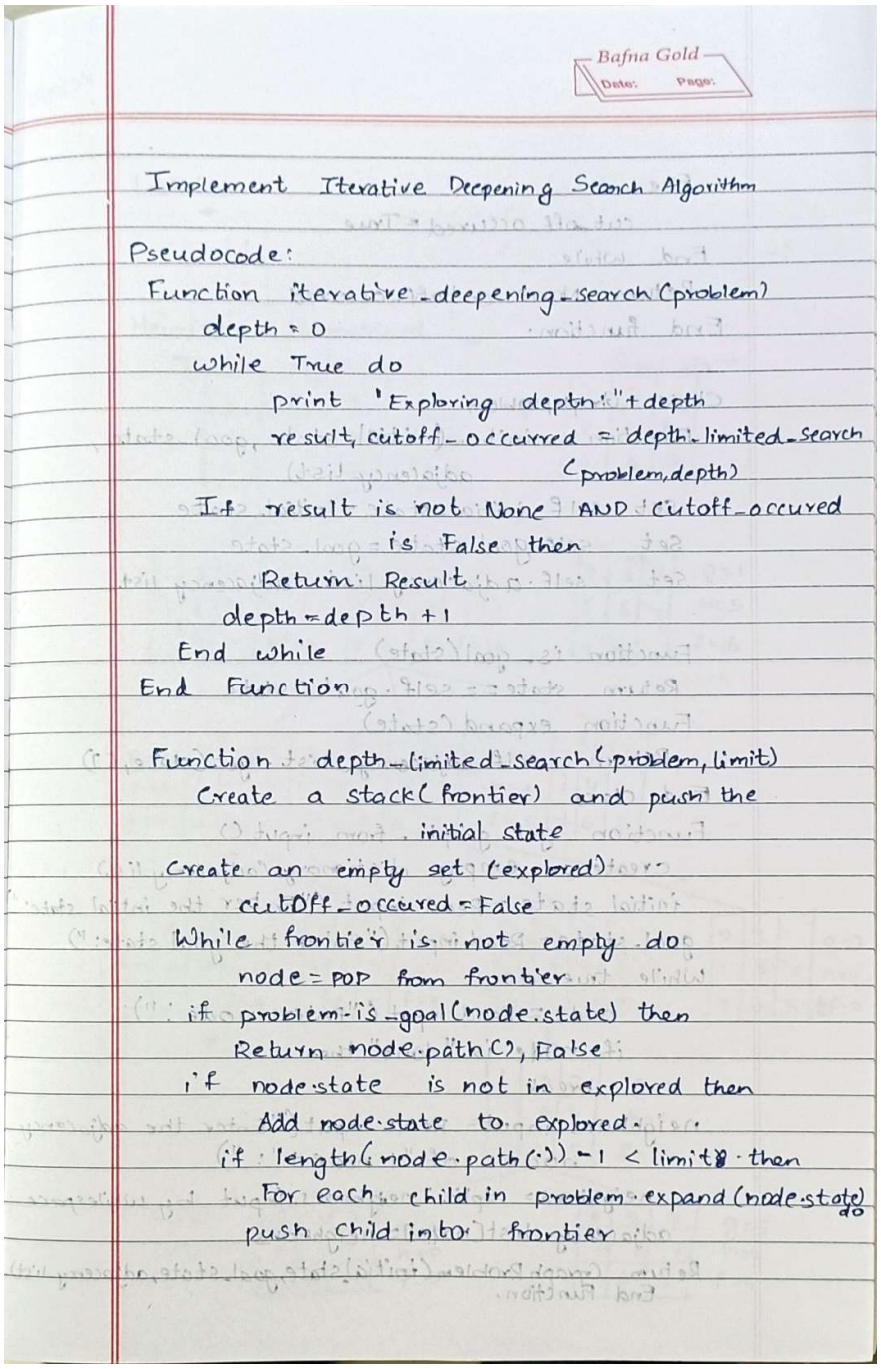
OUTPUT

```
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]
```

LABORATORY PROGRAM – 3

IMPLEMENT ITERATIVE DEEPENING SEARCH ALGORITHM

PSEUDOCODE OR ALGORITHM



Bafna Gold
Date: Page:

Implement Iterative Deepening Search Algorithm

Pseudocode:

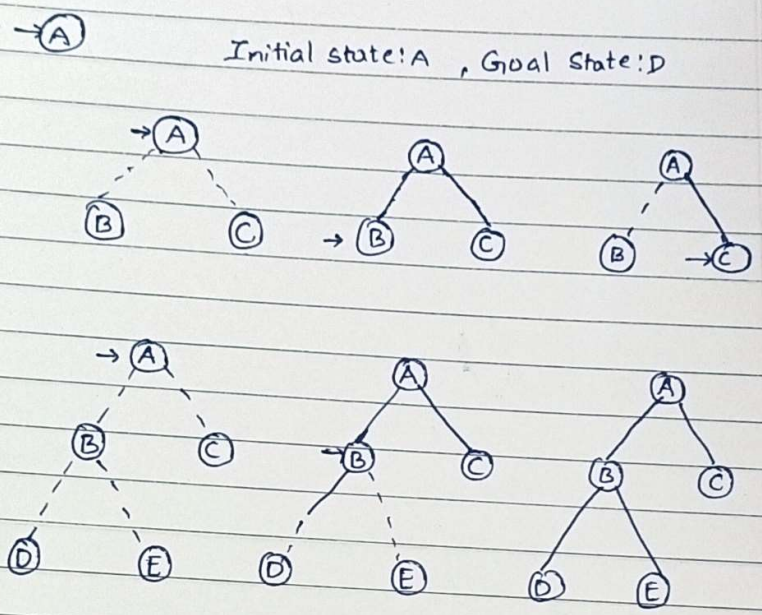
```
Function iterative_deepening_search(problem)
    depth = 0
    while True do
        print 'Exploring depth:' + depth
        result, cutoff_occurred = depth_limited_search
                                  (problem, depth)
    If result is not None AND cutoff_occured
            is False then
        Return Result
    depth = depth + 1
End while
End Function

Function depth_limited_search(problem, limit)
    Create a stack(frontier) and push the
            initial state
Create an empty set (explored)
    cutOff_occured = False
While frontier is not empty do
    node = POP from frontier
if problem.is_goal(node.state) then
    Return node.path(), False
if node.state is not in explored then
    Add node.state to explored.
    if length(node.path()) - 1 < limit then
        For each child in problem.expand(node.state)
        push child into frontier
```

15/10/24

```
    Else
        cut_off_occured = True
    End while
    Return None, cutoff_occured
End function.

Class GraphProblem
Function __init__(initial_state, goal_state,
                  adjacency_list)
    Set self.initial_state = initial_state
    Set self.goal_state = goal_state
    Set self.adjacency_list = adjacency list

    Function is_goal(state)
    Return state == self.goal_state
    Function expand (state)
    Return self.adjacency_list.get(state, [])
End class
Function get_graph_from input ()
    create an empty dictionary (adjacency_list)
    initial_state = Read input ("Enter the intial state:")
    goal_state = Read input ("Enter the goal state:")
    While true do
        node = Read input ("Enter node :").
        if node is "done" then
        break
    neighbors_input = Read input("Enter the adjacency
                  nodes of "+node+": ")
        neighbors = Split : neighbors_input by whilespace
        adjacency_list[node] = neighbors
    Return GraphProblem(initial_state, goal_state, adjacency_list)
    End Function.
```

Bafna Gold
Date: Page:

State Space Tree for Iterative Deepening Search



Goal Reached

Solution path: [A, B, D]

CODE

```
class Node:
    def __init__(self, state, parent=None):
        self.state = state
        self.parent = parent

    def path(self):
        result = []
        current_node = self
        while current_node:
            result.append(current_node.state)
            current_node = current_node.parent
        return result[::-1]


def iterative_deepening_search(problem):
    depth = 0
    while True:
        print(f"Exploring depth: {depth}")
        result, cutoff_occurred = depth_limited_search(problem, depth)
        if result is not None and not cutoff_occurred:
            return result
        depth += 1


def depth_limited_search(problem, limit):
    frontier = [Node(problem.initial_state)]
    explored = set()
    cutoff_occurred = False

    while frontier:
        node = frontier.pop()

        if problem.is_goal(node.state):
            return node.path(), False

        if node.state not in explored:
            explored.add(node.state)
            if len(node.path()) - 1 < limit:
                for child in problem.expand(node.state):
                    frontier.append(Node(child, node))
            else:
                cutoff_occurred = True
    return None, cutoff_occurred
```

```python
class GraphProblem:
    def __init__(self, initial_state, goal_state, adjacency_list):
        self.initial_state = initial_state
        self.goal_state = goal_state
        self.adjacency_list = adjacency_list

    def is_goal(self, state):
        return state == self.goal_state

    def expand(self, state):
        return self.adjacency_list.get(state, [])


def get_graph_from_input():
    adjacency_list = {}
    initial_state = input("Enter the initial state: ").strip()
    goal_state = input("Enter the goal state: ").strip()
    print("Enter the adjacency list for the graph ")
    print("Enter 'done' when finished.")
    while True:
        node = input("Enter node (or 'done' to stop): ").strip()
        if node.lower() == 'done':
            break
        neighbors_input = input(f"Enter the adjacent nodes of {node}: ").strip()
        neighbors = neighbors_input.split()
        adjacency_list[node] = [neighbor.strip() for neighbor in neighbors]

    return GraphProblem(initial_state, goal_state, adjacency_list)
if __name__ == "__main__":
    problem = get_graph_from_input()
    solution = iterative_deepening_search(problem)
    if solution:
        print("Solution Path:", solution)
    else:
        print("No solution found.")
```

OUTPUT

```
Enter the initial state: A
Enter the goal state: F
Enter the adjacency list for the graph
Enter 'done' when finished.
Enter node (or 'done' to stop): A
Enter the adjacent nodes of A: B C
Enter node (or 'done' to stop): B
Enter the adjacent nodes of B: D E
Enter node (or 'done' to stop): C
Enter the adjacent nodes of C: F
Enter node (or 'done' to stop): done
Exploring depth: 0
Exploring depth: 1
Exploring depth: 2
Solution Path: ['A', 'C', 'F']
```

LABORATORY PROGRAM – 4

IMPLEMENT VACUUM CLEANER AGENT

PSEUDOCODE OR ALGORITHM

* Implement Vacuum cleaner agent

Algorithm:

Step-1: Define a function vacuum_agent that takes parameters as location and status.

Step-2: Now check the status:

If the status is "Dirty", it will return "Suck"

Else, if the status is "Clean", it will return "No Action".

Step-3: Now, give the location.

If the location is "P", it will return "left".

If the location is "Q", it will return "right".

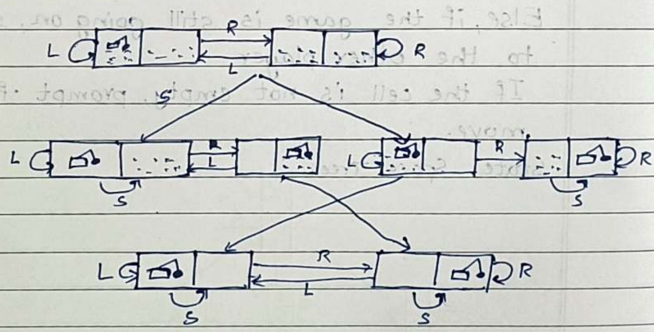
Step-4: Prompt the user to enter a location

If the input is exit, break the loop.

Prompt the user to enter the status.

Step-5: Print the location, status and Action.

State Space Tree:



CODE

```python
def vacuum_agent(location, status):
    if status == "Dirty":
        return "Suck"
    elif status == "Clean":
        return "No Action"

    if location == "P":
        return "left"
    elif location == "Q":
        return "right"
    else:
        return "Invalid Location"

while True:
    location = input("Enter the location (P or Q, or 'exit' to stop): ")
    if location.lower() == 'exit':
        break
    status = input("Enter the status (Dirty or Clean): ")

    action = vacuum_agent(location, status)
    print(f"Location: {location}, Status: {status}, Action: {action}")
```

OUTPUT

```
Enter the location (P or Q, or 'exit' to stop): P
Enter the status (Dirty or Clean): Dirty
Location: P, Status: Dirty, Action: Suck
Enter the location (P or Q, or 'exit' to stop): Q
Enter the status (Dirty or Clean): Dirty
Location: Q, Status: Dirty, Action: Suck
Enter the location (P or Q, or 'exit' to stop): P
Enter the status (Dirty or Clean): Clean
Location: P, Status: Clean, Action: No Action
Enter the location (P or Q, or 'exit' to stop): Q
Enter the status (Dirty or Clean): Clean
Location: Q, Status: Clean, Action: No Action
```