MLA0105 - Artificial Intelligence and Expert System for Gaming

EXPERIMENT- 1: SLIDING PUZZLE (3 X 3)

```
from collections import deque
def print_puzzle(state):
    for i in range(0, 9, 3):
        print(state[i:i+3])
    print()
def get_next_states(state):
    moves = []
    idx = state.index(0)   # empty space (0)
    directions = {
        0: [1, 3],
        1: [0, 2, 4],
        2: [1, 5],
        3: [0, 4, 6],
        4: [1, 3, 5, 7],
        5: [2, 4, 8],
        6: [3, 7],
        7: [4, 6, 8],
        8: [5, 7]
    }

    for d in directions[idx]:
        new_state = state[:]
        new_state[idx], new_state[d] = new_state[d], new_state[idx]
        moves.append(new_state)
```

```
    return moves
def solve_puzzle(start, goal):
    queue = deque([start])
    visited = set()
    parent = {tuple(start): None}

    while queue:
        curr = queue.popleft()

        if curr == goal:
            path = []
            while curr:
                path.append(curr)
                curr = parent[tuple(curr)]
            return path[::-1]

        visited.add(tuple(curr))

        for nxt in get_next_states(curr):
            if tuple(nxt) not in visited:
                parent[tuple(nxt)] = curr
                queue.append(nxt)

    return None

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

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

solution = solve_puzzle(start_state, goal_state)

print("SOLUTION STEPS:\n")
for step in solution:
    print_puzzle(step)
```

Output

```
SOLUTION STEPS:

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

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

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

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

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

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

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

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

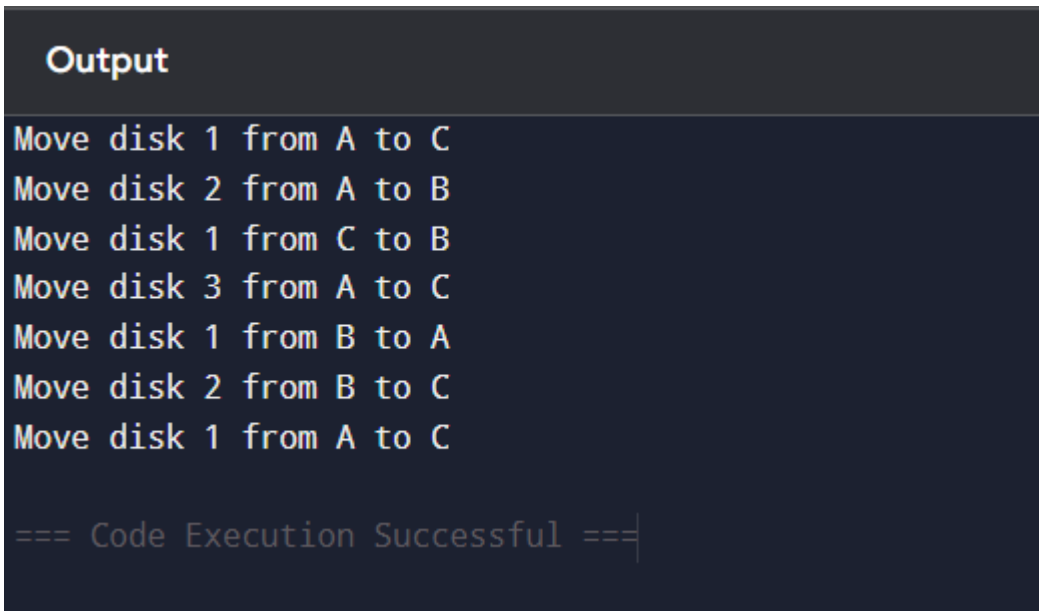
Output

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

EXPERIMENT- 2: Towers of Hanoi problem

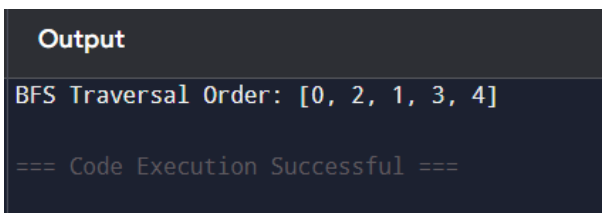
```
def towers_of_hanoi(n, source, temp, destination):
    if n == 1:
        print(f"Move disk 1 from {source} to {destination}")
        return

    towers_of_hanoi(n - 1, source, destination, temp)
    print(f"Move disk {n} from {source} to {destination}")
    towers_of_hanoi(n - 1, temp, source, destination)

n = 3
towers_of_hanoi(n, 'A', 'B', 'C')
```



Experiment-3:-Breadth-First Search (BFS)

```
from collections import deque
graph = {
    0: [2, 1, 3],
    1: [],
    2: [4],
    3: [],
    4: []
}
def bfs(start):
    visited = set()
    queue = deque([start])
    order = []
    while queue:
        node = queue.popleft()
        if node not in visited:
            visited.add(node)
            order.append(node)
            for neighbor in graph[node]:
                queue.append(neighbor)
    return order
result = bfs(0)
print("BFS Traversal Order:", result)
```

Output:

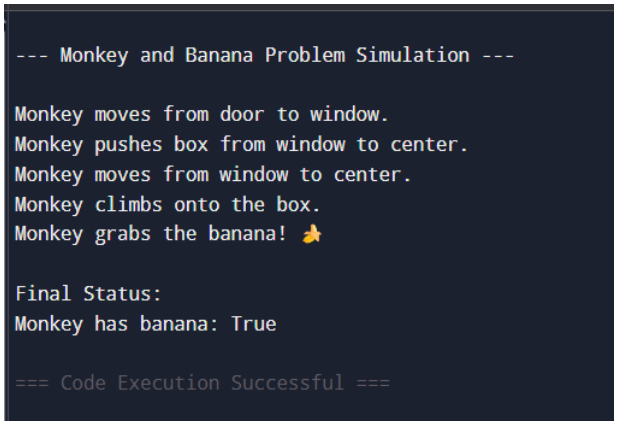


Experiment-4: Monkey and Banana problem

```
class MonkeyBanana:
    def __init__(self):
        self.monkey_position = "door"
        self.box_position = "window"
        self.banana_position = "center"
        self.monkey_on_box = False
        self.has_banana = False
    def move_monkey(self, location):
        print(f"Monkey moves from {self.monkey_position} to {location}.")
        self.monkey_position = location
    def push_box(self, location):
        if self.monkey_position == self.box_position:
            print(f"Monkey pushes box from {self.box_position} to {location}.")
            self.box_position = location
        else:
            print("Monkey must be at the box to push it!")
    def climb_box(self):
        if self.monkey_position == self.box_position:
            print("Monkey climbs onto the box.")
            self.monkey_on_box = True
        else:
            print("Monkey must move to the box first.")
    def grab_banana(self):
        if self.monkey_on_box and self.box_position == self.banana_position:
            print("Monkey grabs the banana! 🍌")
            self.has_banana = True
        else:
```

```
            print("Monkey cannot reach the banana.")
    def solve(self):
        print("\n--- Monkey and Banana Problem Simulation ---\n")
        self.move_monkey("window")
        self.push_box("center")
        self.move_monkey("center")
        self.climb_box()
        self.grab_banana()
        print("\nFinal Status:")
        print(f"Monkey has banana: {self.has_banana}")
problem = MonkeyBanana()
problem.solve()
```

Output:



Experiment-5: place eight queens on an 8 x 8

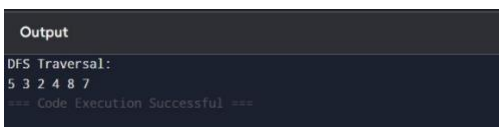
```
def is_safe(board, row, col):
    for i in range(row):
        if board[i] == col or abs(board[i] - col) == abs(i - row):
            return False
    return True


def solve(board, row):
    if row == 8:
        return True
    for col in range(8):
        if is_safe(board, row, col):
            board[row] = col
            if solve(board, row + 1):
                return True
    return False


def print_board(board):
    for r in range(8):
        print(" ".join("Q" if board[r] == c else "." for c in range(8)))


board = [-1] * 8
solve(board, 0)
print_board(board)
```

Output:

Experiment- 6: Depth-First Search (DFS)

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    print(start, end=" ")
    visited.add(start)

    for neighbour in graph[start]:
        if neighbour not in visited:
            dfs(graph, neighbour, visited)

# Graph (adjacency list)
graph = {
    5: [3, 7],
    3: [2, 4],
    7: [8],
    2: [],
    4: [8],
    8: []
}

print("DFS Traversal:")
dfs(graph, 5)
```

Output:



Experiment-7: Water Jug problem

```
from collections import deque


def solve_water_jug():
    visited = set()
    queue = deque([(0, 0)])   # (4-gal jug, 3-gal jug)

    while queue:
        a, b = queue.popleft()

        if (a, b) in visited:
            continue
        visited.add((a, b))

        print(a, b)   # show steps

        if a == 2:    # goal: 2 gallons in 4-gallon jug
            print("Reached 2 gallons in the 4-gallon jug!")
            return

        # Possible actions
        moves = [
            (4, b),       # fill 4-gal
            (a, 3),       # fill 3-gal
            (0, b),       # empty 4-gal
            (a, 0),       # empty 3-gal
            (a - min(a, 3 - b), b + min(a, 3 - b)),  # pour 4 → 3
            (a + min(b, 4 - a), b - min(b, 4 - a))   # pour 3 → 4
```

```
        ]

        for m in moves:
            if m not in visited:
                queue.append(m)

solve_water_jug()
```

Output: