Devesh Vengurlekar
Roll No: 9766
TE Comps A

AI Experiment 4

Aim : Use BFS problem solving method for
a) Water Jug Problem
b) Missionaries & Cannibals

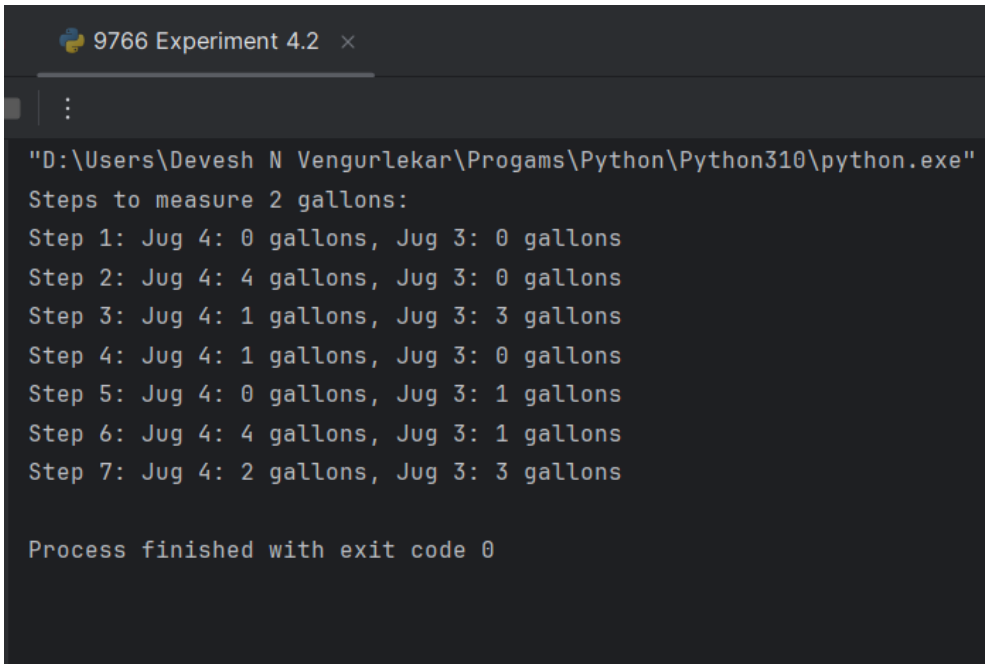
Program:

a) Water Jug Problem

```
# Devesh Vengurlekar
# Roll No: 9766
# TE Comps A

from collections import deque

def pour_water(state, action):
    x, y = state
    if action == 'fill_4':
        return (4, y)
    elif action == 'fill_3':
        return (x, 3)
    elif action == 'empty_4':
        return (0, y)
    elif action == 'empty_3':
        return (x, 0)
    elif action == 'pour_4_to_3':
        amount = min(x, 3 - y)
        return (x - amount, y + amount)
    elif action == 'pour_3_to_4':
        amount = min(y, 4 - x)
        return (x + amount, y - amount)
    else:
        return state

def bfs(initial_state):
    visited = set()
```

```
    queue = deque([([initial_state], initial_state)])
    while queue:
        path, state = queue.popleft()
        if state[0] == 2:
            return path
        visited.add(state)
        for action in ['fill_4', 'fill_3', 'empty_4', 'empty_3', 'pour_4_to_3', 'pour_3_to_4']:
            new_state = pour_water(state, action)
            if new_state not in visited:
                queue.append((path + [new_state], new_state))
    return None

def print_steps(path):
    for i, state in enumerate(path):
        jug_4, jug_3 = state
        print(f"Step {i+1}: Jug 4: {jug_4} gallons, Jug 3: {jug_3} gallons")

initial_state = (0, 0)
path = bfs(initial_state)
if path:
    print("Steps to measure 2 gallons:")
    print_steps(path)
else:
    print("No solution found.")
```

Output:



b) Missionaries & Cannibals

```
# Devesh Vengurlekar
# Roll No: 9766
# TE Comps A

from collections import deque

class State:
    def __init__(self, missionaries_left, cannibals_left, boat_left, missionaries_right,
cannibals_right):
        self.missionaries_left = missionaries_left
        self.cannibals_left = cannibals_left
        self.boat_left = boat_left
        self.missionaries_right = missionaries_right
        self.cannibals_right = cannibals_right

    def is_valid(self):
        if (0 <= self.missionaries_left <= 3 and 0 <= self.cannibals_left <= 3 and
            0 <= self.missionaries_right <= 3 and 0 <= self.cannibals_right <= 3):
            if (self.missionaries_left >= self.cannibals_left or self.missionaries_left == 0) and \
                (self.missionaries_right >= self.cannibals_right or self.missionaries_right == 0):
                return True
        return False

    def is_goal(self):
        return self.missionaries_left == 0 and self.cannibals_left == 0

    def __eq__(self, other):
        return (self.missionaries_left == other.missionaries_left and
                self.cannibals_left == other.cannibals_left and
                self.boat_left == other.boat_left and
                self.missionaries_right == other.missionaries_right and
                self.cannibals_right == other.cannibals_right)

    def __hash__(self):
        return hash((self.missionaries_left, self.cannibals_left, self.boat_left,
                     self.missionaries_right, self.cannibals_right))

def generate_next_states(current_state):
    next_states = []
    moves = [(1, 0), (2, 0), (0, 1), (0, 2), (1, 1)]
```

```
    for m, c in moves:
        if current_state.boat_left:
            new_state = State(current_state.missionaries_left - m,
                              current_state.cannibals_left - c,
                              1 - current_state.boat_left,
                              current_state.missionaries_right + m,
                              current_state.cannibals_right + c)
        else:
            new_state = State(current_state.missionaries_left + m,
                              current_state.cannibals_left + c,
                              1 - current_state.boat_left,
                              current_state.missionaries_right - m,
                              current_state.cannibals_right - c)
        if new_state.is_valid():
            next_states.append(new_state)
    return next_states

def bfs_search():
    start_state = State(3, 3, 1, 0, 0)
    goal_state = State(0, 0, 0, 3, 3)
    queue = deque([(start_state, [])])
    visited = set()
    while queue:
        current_state, path = queue.popleft()
        if current_state.is_goal():
            return path
        if current_state not in visited:
            visited.add(current_state)
            next_states = generate_next_states(current_state)
            for next_state in next_states:
                if next_state not in visited:
                    queue.append((next_state, path + [current_state]))
    return None

def print_state_description(state):
    left_shore = f"{state.missionaries_left} Missionaries and {state.cannibals_left} Cannibals
on the Left Shore"
    right_shore = f"{state.missionaries_right} Missionaries and {state.cannibals_right}
Cannibals on the Right Shore"
    print(f"{left_shore}, {right_shore}\n")

if __name__ == "__main__":
    solution_path = bfs_search()
    if solution_path:
```

```
    print("Solution Path:")
    for i, state in enumerate(solution_path):
        print(f"Step {i + 1}:")
        print_state_description(state)
else:
    print("No solution found.")
```

Output: