AI LAB 4 – 9763-Harsh Parmar– Batch D

Water Jug problem using BFS :

Code:

```
from collections import deque


class State:
    def__init__(self, jugs):
        self.jugs = jugs


    def__eq__(self, other):
        return self.jugs == other.jugs


    def__hash__(self):
        return hash(tuple(self.jugs))


def successors(state, jug_sizes):
    successors = []
    for i in range(len(state.jugs)):
        for j in range(len(state.jugs)):
            if i != j:
                pour_amount = min(state.jugs[i], jug_sizes[j] - state.jugs[j])
                if pour_amount > 0:
                    new_jugs = list(state.jugs)
                    new_jugs[i] -= pour_amount
```

```
                new_jugs[j] += pour_amount
                successors.append(State(tuple(new_jugs)))
    return successors

def bfs(initial_state, goal, jug_sizes):
    queue = deque([(initial_state, [])])
    visited = set()

    while queue:
        state, actions = queue.popleft()
        if state == goal:
            return actions
        if state not in visited:
            visited.add(state)
            for successor in successors(state, jug_sizes):
                queue.append((successor, actions + [successor]))

    return None

def main():
    jug_sizes = (5, 3)  # Jug sizes (e.g., (5, 3) represents jugs of size 5 and 3)
    initial_state = State((0, 0))  # Initial state of the jugs
    goal_state = State((4, 0))  # Goal state to reach

    solution = bfs(initial_state, goal_state, jug_sizes)
    if solution:
```

```
        print("Solution:")
        for action in solution:
            print(action)
    else:
        print("No solution found.")

if __name__ == "__main__":
    main()
```

<u>Missionaries and Cannibals :</u>

Code:

```
from collections import deque

class State:
    def __init__(self, missionaries, cannibals, boat):
        self.missionaries = missionaries
        self.cannibals = cannibals
        self.boat = boat

    def __eq__(self, other):
        return self.missionaries == other.missionaries and self.cannibals ==
other.cannibals and self.boat == other.boat

    def __hash__(self):
        return hash((self.missionaries, self.cannibals, self.boat))
```

```
def successors(state):
    successors = []
    if state.boat == 'left':
        for m in range(3):
            for c in range(3):
                if 1 <= m + c <= 2:
                    new_state = State(state.missionaries - m, state.cannibals - c, 'right')
                    if 0 <= new_state.missionaries <= 3 and 0 <= new_state.cannibals <= 3 and (new_state.missionaries >= new_state.cannibals or new_state.missionaries == 0) and ((3 - new_state.missionaries) >= (3 - new_state.cannibals) or new_state.missionaries == 3):
                        successors.append(new_state)
    else:
        for m in range(3):
            for c in range(3):
                if 1 <= m + c <= 2:
                    new_state = State(state.missionaries + m, state.cannibals + c, 'left')
                    if 0 <= new_state.missionaries <= 3 and 0 <= new_state.cannibals <= 3 and (new_state.missionaries >= new_state.cannibals or new_state.missionaries == 0) and ((3 - new_state.missionaries) >= (3 - new_state.cannibals) or new_state.missionaries == 3):
                        successors.append(new_state)
    return successors


def bfs(initial_state, goal_state):
    queue = deque([(initial_state, [])])
    visited = set()
```

```
    while queue:
        state, actions = queue.popleft()
        if state == goal_state:
            return actions
        if state not in visited:
            visited.add(state)
            for successor in successors(state):
                queue.append((successor, actions + [successor]))

    return None

def main():
    initial_state = State(3, 3, 'left')  # Initial state of the missionaries and
cannibals
    goal_state = State(0, 0, 'right')  # Goal state to reach

    solution = bfs(initial_state, goal_state)
    if solution:
        print("Solution:")
        for action in solution:
            print(f"Move {action.missionaries} missionaries and {action.cannibals}
cannibals to the {action.boat}.")
    else:
        print("No solution found.")

if___name___== "__main__":
```

```
    main()
```

OUTPUT:

```
Move 3 missionaries and 1 cannibals to the right.
Move 3 missionaries and 2 cannibals to the left.
Move 3 missionaries and 0 cannibals to the right.
Move 3 missionaries and 1 cannibals to the left.
Move 1 missionaries and 1 cannibals to the right.
Move 2 missionaries and 2 cannibals to the left.
Move 0 missionaries and 2 cannibals to the right.
Move 0 missionaries and 3 cannibals to the left.
Move 0 missionaries and 1 cannibals to the right.
Move 0 missionaries and 2 cannibals to the left.
Move 0 missionaries and 0 cannibals to the right.
PS C:\Users\gaura\Desktop\Pracs\AI>
```

POSTLAB:

AI Lab-3 Harsh Parmar 9763

The time complexity of Water Jug Problem depends on th algorithm used to solve it. Generally when solving the water Jug Problem Using a brute ~~fro~~ force approach, the time complexity can be exponential in worst case. This is because the search space grown expoentially with the number of actions required to reach the goal state. Therefore the time complexity can be expressed as ($O(b^d)$) b is braching factor & d is the depth of a search tree.

DFS is not usually used for solving the Water Jug Problem because it does not guarantee finding the shortest path to the goal state. DFS explores the search space depth first meaning it explores one branch of the search tree has far as possible before backtracking. While DFS may find a solution for the Water Jug Problem, it may not find the optimal solution. Addition DFS may get stuck in infinite loops if the search space contains cycles. Therefore BFS or A* are used for Water Jug Problem.