Department of Computer Engineering
Academic Term II : 23-24

Class: B.E (Computer), Sem – VI **Subject Name: Artificial Intelligence**

Student Name: Jatin Jaywant Kadu **Roll No: 9548**

Practical No:	4
Title:	Solve by implementing BFS method in Python :- a) Missionaíies & cannibals b) Wateí Jug Píoblem
Date of Performance:	
Date of Submission:	

Rubrics for Evaluation:

Sr. No	Performance Indicator	Excellent	Good	Below Average	Marks
1	On time Completion & Submission (01)	01 (On Time)	NA	00 (Not on Time)	
2	Logic/Algorithm Complexity analysis(03)	03(Correct)	02(Partial)	01 (Tried)	
3	Coding Standards (03): Comments/indention/Naming conventions Test Cases /Output	03(All used)	02 (Partial)	01 (rarely followed)	
4	Post Lab Assignment (03)	03(done well)	2 (Partially Correct)	1(submitted)	
Total					

Signature of the Teacher:

Experiment No: 4

Title: Use BFS problem solving method for

a) Water Jug Problem
b) Missionaries & Cannibals

Objective: To write programs which solve the water jug problem and Missionaries & Cannibals problem in an efficient manner using Breadth First Search.

Theory:

Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes. Then, it selects the nearest node and explores all the unexplored nodes. While using BFS for traversal, any node in the graph can be considered as the root node.

WATER JUG PROBLEM:

Given a 'm' liter jug and a 'n' liter jug, both the jugs are initially empty. The jugs don't have markings to allow measuring smaller quantities. You have to use the jugs to measure d liters of water where d is less than n.

(X, Y) corresponds to a state where X refers to amount of water in Jug1 and Y refers to amount of water in Jug2

Determine the path from initial state (xi, yi) to final state (xf, yf), where (xi, yi) is (0, 0) which indicates both Jugs are initially empty and (xf, yf) indicates a state which could be (0, d) or (d, 0).

The operations you can perform are:

4. Empty a Jug, (X, Y)->(0, Y) Empty Jug 1
5. Fill a Jug, (0, 0)-> (X, 0) Fill Jug 1
6. Pour water from one jug to the other until one of the jugs is either empty or full, (X, Y) -> (X-d, Y+d)

Algorithm:

Initialize a queue to implement BFS.
Since, initially, both the jugs are empty, insert the state {0, 0} into the queue.
Perform the following state, till the queue becomes empty:

1. Pop out the first element of the queue.
2. If the value of popped element is equal to Z, return True.
3. Let X_left and Y_left be the amount of water left in the jugs respectively.
4. Now perform the fill operation:
    a) If the value of X_left < X, insert ({X_left, Y}) into the HashMap, since this state hasn't been visited and some water can still be poured in the jug.
    b) If the value of Y_left < Y, insert ({Y_left, X}) into the HashMap, since this state hasn't been visited and some water can still be poured in the jug.
5. Perform the empty operation:
    a. If the state ({0, Y_left}) isn't visited, insert it into the HashMap, since we can empty any of the jugs.
    b. Similarly, if the state ({X_left, 0) isn't visited, insert it into the HashMap, since we can empty any of the jugs.
6. Perform the transfer of water operation:
    a. min ({X-X_left, Y}) can be poured from second jug to first jug. Therefore, in case – {X + min ({X-X_left, Y}), Y – min ({X-X_left, Y}) isn't visited, put it into a HashMap.
    b. min ({X_left, Y-Y_left}) can be poured from first jug to second jug. Therefore, in case – {X_left – min ({X_left, Y – X_left}), Y + min ({X_left, Y – Y_left}) isn't visited, put it into a HashMap.
7. Return False, since, it is not possible to measure Z liters.

MISSIONARIES AND CANNIBALS' PROBLEM:

In this problem, three missionaries and three cannibals must cross a river using a boat which can carry at most two people, under the constraint that, for both banks, that the missionaries present on the bank cannot be outnumbered by cannibals. The boat cannot cross the river by itself with no people on board.

A system for solving the Missionaries and Cannibals problem whereby the current state is represented by a simple vector ⟨m, c, b⟩. The vector's elements represent the number of missionaries, cannibals, and whether the boat is on the wrong side, respectively. Since the boat and all of the missionaries and cannibals start on the wrong side, the vector is initialized to ⟨3,3,1⟩. Actions are represented using vector subtraction/addition to manipulate the state vector.
For instance, if a lone cannibal crossed the river, the vector ⟨0,1,1⟩ would be subtracted from the state to yield ⟨3,2,0⟩. The state would reflect that there are still three missionaries and two cannibals on the wrong side, and that the boat is now on the opposite bank.

To fully solve the problem, a simple tree is formed with the initial state as the root. The five possible actions (⟨1,0,1⟩, ⟨2,0,1⟩, ⟨0,1,1⟩, ⟨0,2,1⟩, and ⟨1,1,1⟩) are then *subtracted* from the initial state, with the result forming children nodes of the root. Any node that has more cannibals than missionaries on either bank is in an invalid state, and is therefore removed from further consideration.

The valid children nodes generated would be ⟨3,2,0⟩, ⟨3,1,0⟩, and ⟨2,2,0⟩. For each of these remaining nodes, children nodes are generated by *adding* each of the possible action vectors. The algorithm continues alternating subtraction and addition for each level of the tree until a node is generated with the vector ⟨0,0,0⟩ as its value. This is the goal state, and the path from the root of the tree to this node represents a sequence of actions that solves the problem.

Code:

```
from collections import deque

def water_jug_bfs(capacity_jug1, capacity_jug2, target):
    visited_states = set()
    queue = deque([(0, 0, "Initial State")])  # Initial state: both jugs are empty
    visited_states.add((0, 0))
    parent = {}  # Dictionary to keep track of the parent state for each state

    while queue:
        current_state = queue.popleft()
        jug1, jug2, action = current_state

        # Check if the goal state is reached
        if jug2 == target:
            print_steps(current_state, parent)
            return

        # Fill jug1
        fill_jug1 = (capacity_jug1, jug2, "Fill Jug1")
        if fill_jug1 not in visited_states:
            visited_states.add(fill_jug1)
            queue.append(fill_jug1)
            parent[fill_jug1] = current_state

        # Fill jug2
        fill_jug2 = (jug1, capacity_jug2, "Fill Jug2")
        if fill_jug2 not in visited_states:
            visited_states.add(fill_jug2)
            queue.append(fill_jug2)
            parent[fill_jug2] = current_state

        # Pour water from jug1 to jug2
        pour_jug1_to_jug2 = (max(0, jug1 - (capacity_jug2 - jug2)), min(jug2 + jug1, capacity_jug2), "Pour
Jug1 to Jug2")
        if pour_jug1_to_jug2 not in visited_states:
            visited_states.add(pour_jug1_to_jug2)
            queue.append(pour_jug1_to_jug2)
            parent[pour_jug1_to_jug2] = current_state
```

```
        # Pour water from jug2 to jug1
        pour_jug2_to_jug1 = (min(jug1 + jug2, capacity_jug1), max(0, jug2 - (capacity_jug1 - jug1)), "Pour
Jug2 to Jug1")
        if pour_jug2_to_jug1 not in visited_states:
            visited_states.add(pour_jug2_to_jug1)
            queue.append(pour_jug2_to_jug1)
            parent[pour_jug2_to_jug1] = current_state

        # Empty jug1
        empty_jug1 = (0, jug2, "Empty Jug1")
        if empty_jug1 not in visited_states:
            visited_states.add(empty_jug1)
            queue.append(empty_jug1)
            parent[empty_jug1] = current_state

        # Empty jug2
        empty_jug2 = (jug1, 0, "Empty Jug2")
        if empty_jug2 not in visited_states:
            visited_states.add(empty_jug2)
            queue.append(empty_jug2)
            parent[empty_jug2] = current_state

def print_steps(state, parent):
    steps = []
    while state[2] != "Initial State":
        steps.append(state)
        state = parent[state]
    steps.append((0, 0, "Initial State"))

    steps.reverse()
    for step in steps:
        print(f"{step[2]}: {step[0]} | {step[1]}")

# Example usage
capacity_jug1 = 3
capacity_jug2 = 4
target = 2

water_jug_bfs(capacity_jug1, capacity_jug2, target)
```

Output:

```
PS C:\Users\hacke\OneDrive\Desktop\SEM VI\AI> & C:/Users/hacke/AppData/Local/Programs/Python/Python311/py
thon.exe "c:/Users/hacke/OneDrive/Desktop/SEM VI/AI/bfs_waterjug.py"
Initial State: 0 | 0
Fill Jug1: 3 | 0
Pour Jug1 to Jug2: 0 | 3
Fill Jug1: 3 | 3
Pour Jug1 to Jug2: 2 | 4
Empty Jug2: 2 | 0
Pour Jug1 to Jug2: 0 | 2
```

Code:

```
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:

```
PS C:\Users\hacke\OneDrive\Desktop\SEM VI\AI> & C:/Users/hacke/AppData/Local/Programs/Python/Python311/python.exe "c:/Users/hacke/OneDrive/Desktop/SEM VI/AI/bfs_cannibalproblem.py"
Solution Path:
Step 1:
3 Missionaries and 3 Cannibals on the Left Shore, 0 Missionaries and 0 Cannibals on the Right Shore

Step 2:
3 Missionaries and 1 Cannibals on the Left Shore, 0 Missionaries and 2 Cannibals on the Right Shore

Step 3:
3 Missionaries and 2 Cannibals on the Left Shore, 0 Missionaries and 1 Cannibals on the Right Shore

Step 4:
3 Missionaries and 0 Cannibals on the Left Shore, 0 Missionaries and 3 Cannibals on the Right Shore

Step 5:
3 Missionaries and 1 Cannibals on the Left Shore, 0 Missionaries and 2 Cannibals on the Right Shore

Step 6:
1 Missionaries and 1 Cannibals on the Left Shore, 2 Missionaries and 2 Cannibals on the Right Shore
```

```
Step 6:
1 Missionaries and 1 Cannibals on the Left Shore, 2 Missionaries and 2 Cannibals on the Right Shore

Step 7:
2 Missionaries and 2 Cannibals on the Left Shore, 1 Missionaries and 1 Cannibals on the Right Shore

Step 8:
0 Missionaries and 2 Cannibals on the Left Shore, 3 Missionaries and 1 Cannibals on the Right Shore

Step 9:
0 Missionaries and 3 Cannibals on the Left Shore, 3 Missionaries and 0 Cannibals on the Right Shore

Step 10:
0 Missionaries and 1 Cannibals on the Left Shore, 3 Missionaries and 2 Cannibals on the Right Shore

Step 11:
1 Missionaries and 1 Cannibals on the Left Shore, 2 Missionaries and 2 Cannibals on the Right Shore
```