Fr. Conceicao Rodrigues College of Engineering Fr. Agnel Ashram, Bandstand, Bandra (W), Mumbai - 400050

Department of Computer Engineering
Academic Term II : 23-24

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

No:

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 cannibalsthan 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.

Water Jug Problem Using BFS

```
from collections import deque

# Define the state of the jugs
class State:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __hash__(self):
        return hash((self.x, self.y))

    def __str__(self):
        return f'({self.x}, {self.y})'

# Function to find all possible next states from the current state
def get_next_states(curr_state, max_x, max_y):
    next_states = []
    # Filling the first jug
    next_states.append(State(max_x, curr_state.y))
    # Filling the second jug
    next_states.append(State(curr_state.x, max_y))
    # Emptying the first jug
    next_states.append(State(0, curr_state.y))
    # Emptying the second jug
    next_states.append(State(curr_state.x, 0))
    # Pouring from first jug to second jug
    x_to_y = min(curr_state.x, max_y - curr_state.y)
    next_states.append(State(curr_state.x - x_to_y, curr_state.y + x_to_y))
    # Pouring from second jug to first jug
    y_to_x = min(max_x - curr_state.x, curr_state.y)
    next_states.append(State(curr_state.x + y_to_x, curr_state.y - y_to_x))
    return [state for state in next_states if state != curr_state]

# Breadth-first search algorithm to find the solution
def bfs(start_state, target_volume_x, target_volume_y):
    max_x = target_volume_x
    max_y = target_volume_y
    visited = set()
    queue = deque([(start_state, [])])

    while queue:
        curr_state, path = queue.popleft()
        if curr_state.x == target_volume_x and curr_state.y == target_volume_y:
            return path
        visited.add(curr_state)
```

```
        next_states = get_next_states(curr_state, max_x, max_y)
        for next_state in next_states:
            if next_state not in visited:
                queue.append((next_state, path + [next_state]))

    return None

# Main function
def main():
    print("Water Jug Problem Solver")
    max_x = int(input("Enter the capacity of the first jug: "))
    max_y = int(input("Enter the capacity of the second jug: "))
    target_volume_x = int(input("Enter the target volume for the first jug: "))
    target_volume_y = int(input("Enter the target volume for the second jug: "))

    start_state = State(0, 0)
    solution = bfs(start_state, target_volume_x, target_volume_y)

    if solution:
        print("Solution found:")
        for i, state in enumerate(solution):
            print(f"Step {i + 1}: Fill jug ({state.x}, {state.y})")
    else:
        print("No solution exists.")

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

Output:

```
Enter the capacity of Jug1:  4
Enter the capacity of Jug2:  3
Enter the target amount:  2
Path from initial state to solution state ::
( 0 , 0 )
( 0 , 3 )
( 4 , 0 )
( 4 , 3 )
( 3 , 0 )
( 1 , 3 )
( 3 , 3 )
( 4 , 2 )
( 0 , 2 )
```

Missionaries and Cannibals Problem using BFS;

```
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 __str__(self):
        return f'M: {self.missionaries}, C: {self.cannibals}, B: {self.boat}'

def is_valid_state(state):
    if state.missionaries < 0 or state.missionaries > 3 or \
       state.cannibals < 0 or state.cannibals > 3 or \
       state.missionaries < state.cannibals and state.missionaries > 0 or \
       state.missionaries > state.cannibals and state.missionaries < 3:
        return False
    return True

def get_next_states(curr_state):
    next_states = []
    if curr_state.boat == 'left':
        next_states.append(State(curr_state.missionaries - 2,
curr_state.cannibals, 'right'))
        next_states.append(State(curr_state.missionaries, curr_state.cannibals
- 2, 'right'))
        next_states.append(State(curr_state.missionaries - 1,
curr_state.cannibals - 1, 'right'))
        next_states.append(State(curr_state.missionaries - 1,
curr_state.cannibals, 'right'))
        next_states.append(State(curr_state.missionaries, curr_state.cannibals
- 1, 'right'))
    else:
        next_states.append(State(curr_state.missionaries + 2,
curr_state.cannibals, 'left'))
        next_states.append(State(curr_state.missionaries, curr_state.cannibals
+ 2, 'left'))
        next_states.append(State(curr_state.missionaries + 1,
curr_state.cannibals + 1, 'left'))
        next_states.append(State(curr_state.missionaries + 1,
curr_state.cannibals, 'left'))
        next_states.append(State(curr_state.missionaries, curr_state.cannibals
+ 1, 'left'))

    return [state for state in next_states if is_valid_state(state)]

def bfs(start_state, target_state):
    visited = set()
```

```python
    queue = deque([(start_state, [])])

    while queue:
        curr_state, path = queue.popleft()
        if curr_state == target_state:
            return path
        visited.add(curr_state)
        next_states = get_next_states(curr_state)
        for next_state in next_states:
            if next_state not in visited:
                queue.append((next_state, path + [next_state]))

    return None

def main():
    print("Missionaries and Cannibals Problem Solver")
    start_state = State(3, 3, 'left')
    target_state = State(0, 0, 'right')
    solution = bfs(start_state, target_state)

    if solution:
        print("Solution found:")
        for i, state in enumerate(solution):
            print(f"Step {i + 1}: {state}")
    else:
        print("No solution exists.")

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

```
PS C:\Users\Royce Dmello\OneDrive\Documents\AI> python 4.2.py
Missionaries and Cannibals Problem Solver
Solution found:
Step 1: M: 3, C: 1, B: right
Step 2: M: 3, C: 2, B: left
Step 3: M: 3, C: 0, B: right
Step 4: M: 3, C: 1, B: left
Step 5: M: 1, C: 1, B: right
Step 6: M: 2, C: 2, B: left
Step 7: M: 0, C: 2, B: right
Step 8: M: 0, C: 3, B: left
Step 9: M: 0, C: 1, B: right
Step 10: M: 1, C: 1, B: left
```


```
Step 8: M: 0, C: 3, B: left
Step 9: M: 0, C: 1, B: right
Step 10: M: 1, C: 1, B: left
```


```
Step 11: M: 0, C: 0, B: right
PS C:\Users\Royce Dmello\OneDrive\Documents\AI>
```