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: Siddhesh Pradhan **Roll No: 9632**

Practical No:	8
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.

<u>WATER JUG PROBLEM</u>:

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.

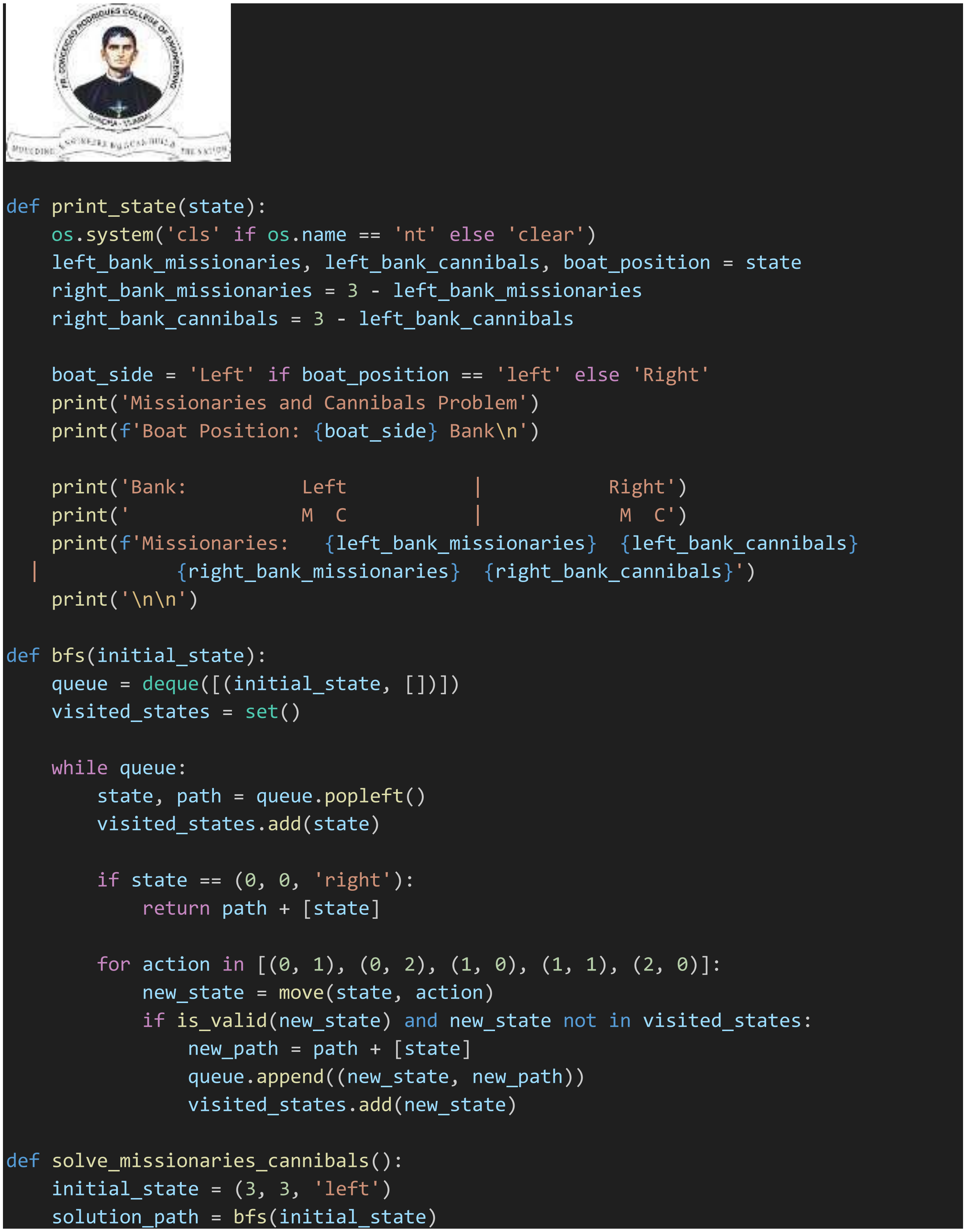
For instance, if a lone cannibal crossed the river, the vector ⟨0,1,1⟩ would be subtracted from the state to yield ⟨3,2,0⟩. The state would reflect that there are still three missionaries and two cannibals on the wrong side, and that the boat is now on the opposite bank.

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.

expt8_MisCanBFS.py

```
import time
import os
from collections import deque

def move(state, action):
    # Unpack the current state
    left_bank_missionaries, left_bank_cannibals, boat_position = state

    # Apply the action based on the boat position
    if boat_position == 'left':
        left_bank_missionaries -= action[0]
        left_bank_cannibals -= action[1]
        boat_position = 'right'
    else:
        left_bank_missionaries += action[0]
        left_bank_cannibals += action[1]
        boat_position = 'left'

    return (left_bank_missionaries, left_bank_cannibals, boat_position)

def is_valid(state):
    left_bank_missionaries, left_bank_cannibals, boat_position = state
    right_bank_missionaries = 3 - left_bank_missionaries
    right_bank_cannibals = 3 - left_bank_cannibals

    # Check constraints
    if left_bank_missionaries < 0 or left_bank_cannibals < 0 or \
        right_bank_missionaries < 0 or right_bank_cannibals < 0:
        return False
    if left_bank_missionaries > 0 and left_bank_missionaries < left_bank_cannibals:
        return False
    if right_bank_missionaries > 0 and right_bank_missionaries <
right_bank_cannibals:
        return False
    return True
```



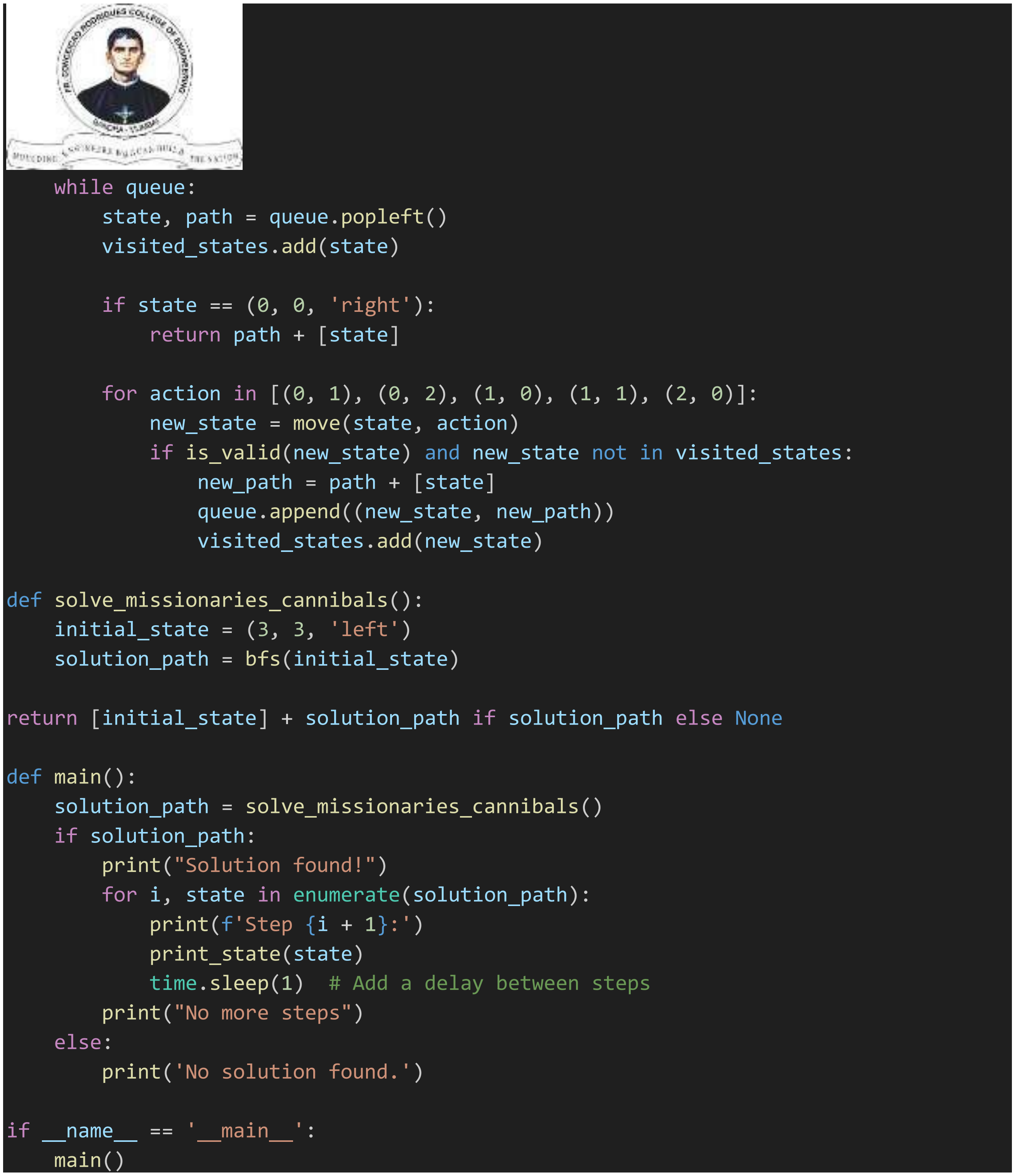
```python
def print_state(state):
    os.system('cls' if os.name == 'nt' else 'clear')
    left_bank_missionaries, left_bank_cannibals, boat_position = state
    right_bank_missionaries = 3 - left_bank_missionaries
    right_bank_cannibals = 3 - left_bank_cannibals

    boat_side = 'Left' if boat_position == 'left' else 'Right'
    print('Missionaries and Cannibals Problem')
    print(f'Boat Position: {boat_side} Bank\n')

    print('Bank:           Left           |           Right')
    print('                M  C           |            M  C')
    print(f'Missionaries:   {left_bank_missionaries}  {left_bank_cannibals}
  |           {right_bank_missionaries}  {right_bank_cannibals}')
    print('\n\n')

def bfs(initial_state):
    queue = deque([(initial_state, [])])
    visited_states = set()

    while queue:
        state, path = queue.popleft()
        visited_states.add(state)

        if state == (0, 0, 'right'):
            return path + [state]

        for action in [(0, 1), (0, 2), (1, 0), (1, 1), (2, 0)]:
            new_state = move(state, action)
            if is_valid(new_state) and new_state not in visited_states:
                new_path = path + [state]
                queue.append((new_state, new_path))
                visited_states.add(new_state)

def solve_missionaries_cannibals():
    initial_state = (3, 3, 'left')
    solution_path = bfs(initial_state)
```

```python
return [initial_state] + solution_path if solution_path else None

def main():
    solution_path = solve_missionaries_cannibals()
    if solution_path:
        print("Solution found!")
        for i, state in enumerate(solution_path):
            print(f'Step {i + 1}:')
            print_state(state)
            time.sleep(1)  # Add a delay between steps
        print("No more steps")
    else:
        print('No solution found.')

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

expt8_MisCanBFS.py

```python
import time
import os
from collections import deque

def move(state, action):
    # Unpack the current state
    left_bank_missionaries, left_bank_cannibals, boat_position = state

    # Apply the action based on the boat position
    if boat_position == 'left':
        left_bank_missionaries -= action[0]
        left_bank_cannibals -= action[1]
        boat_position = 'right'
    else:
        left_bank_missionaries += action[0]
        left_bank_cannibals += action[1]
        boat_position = 'left'

    return (left_bank_missionaries, left_bank_cannibals, boat_position)
```

```
def is_valid(state):
    left_bank_missionaries, left_bank_cannibals, boat_position = state
    right_bank_missionaries = 3 - left_bank_missionaries
    right_bank_cannibals = 3 - left_bank_cannibals

    # Check constraints
    if left_bank_missionaries < 0 or left_bank_cannibals < 0 or \
        right_bank_missionaries < 0 or right_bank_cannibals < 0:
        return False
    if left_bank_missionaries > 0 and left_bank_missionaries < left_bank_cannibals:
        return False
    if right_bank_missionaries > 0 and right_bank_missionaries <
right_bank_cannibals:
        return False
    return True




def print_state(state):
    os.system('cls' if os.name == 'nt' else 'clear')
    left_bank_missionaries, left_bank_cannibals, boat_position = state
    right_bank_missionaries = 3 - left_bank_missionaries
    right_bank_cannibals = 3 - left_bank_cannibals

    boat_side = 'Left' if boat_position == 'left' else 'Right'
    print('Missionaries and Cannibals Problem')
    print(f'Boat Position: {boat_side} Bank\n')

    print('Bank:          Left           |           Right')
    print('               M  C           |            M  C')
    print(f'Missionaries:   {left_bank_missionaries}  {left_bank_cannibals}
  |            {right_bank_missionaries}  {right_bank_cannibals}')
    print('\n\n')

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



```
    while queue:
        state, path = queue.popleft()
        visited_states.add(state)

        if state == (0, 0, 'right'):
            return path + [state]

        for action in [(0, 1), (0, 2), (1, 0), (1, 1), (2, 0)]:
            new_state = move(state, action)
            if is_valid(new_state) and new_state not in visited_states:
                new_path = path + [state]
                queue.append((new_state, new_path))
                visited_states.add(new_state)

def solve_missionaries_cannibals():
    initial_state = (3, 3, 'left')
    solution_path = bfs(initial_state)

return [initial_state] + solution_path if solution_path else None

def main():
    solution_path = solve_missionaries_cannibals()
    if solution_path:
        print("Solution found!")
        for i, state in enumerate(solution_path):
            print(f'Step {i + 1}:')
            print_state(state)
            time.sleep(1)  # Add a delay between steps
        print("No more steps")
    else:
        print('No solution found.')

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