1. **Implement and Demonstrate Best First Search Algorithm on Missionaries-Cannibals Problems using Python**

from collections import deque

# Define the initial state

initial\_state = {'left': (3, 3), 'right': (0, 0), 'boat': 'left'}

# Define the goal state

goal\_state = {'left': (0, 0), 'right': (3, 3), 'boat': 'right'}

# Define a function to check if a state is valid

def is\_valid(state):

left\_m, left\_c = state['left']

right\_m, right\_c = state['right']

if left\_m < 0 or left\_c < 0 or right\_m < 0 or right\_c < 0:

return False

if left\_m > 3 or left\_c > 3 or right\_m > 3 or right\_c > 3:

return False

if left\_m < left\_c and left\_m > 0:

return False

if right\_m < right\_c and right\_m > 0:

return False

return True

# Define a function to generate all possible next states from the current state

def generate\_next\_states(current\_state):

next\_states = []

for i in range(3):

for j in range(3):

if i + j > 2 or i + j == 0:

continue

if current\_state['boat'] == 'left':

new\_state = {

'left': (current\_state['left'][0] - i, current\_state['left'][1] - j),

'right': (current\_state['right'][0] + i, current\_state['right'][1] + j),

'boat': 'right'

}

else:

new\_state = {

'left': (current\_state['left'][0] + i, current\_state['left'][1] + j),

'right': (current\_state['right'][0] - i, current\_state['right'][1] - j),

'boat': 'left'

}

if is\_valid(new\_state):

next\_states.append(new\_state)

return next\_states

# Define the breadth-first search function

def bfs(initial\_state, goal\_state):

visited = set()

queue = deque([(initial\_state, [])])

while queue:

current\_state, path = queue.popleft()

if current\_state == goal\_state:

return path

if tuple(current\_state['left'] + current\_state['right'] + (current\_state['boat'],)) in visited:

continue

visited.add(tuple(current\_state['left'] + current\_state['right'] + (current\_state['boat'],)))

for next\_state in generate\_next\_states(current\_state):

queue.append((next\_state, path + [next\_state]))

return None

# Find the solution using BFS

solution = bfs(initial\_state, goal\_state)

# Print the solution

if solution:

print("Solution found with", len(solution), "steps:")

for i, state in enumerate(solution):

print("Step", i + 1, ":", state)

else:

print("No solution found.")

**OUTPUT**

Solution found with 11 steps:

Step 1 : {'left': (3, 1), 'right': (0, 2), 'boat': 'right'}

Step 2 : {'left': (3, 2), 'right': (0, 1), 'boat': 'left'}

Step 3 : {'left': (3, 0), 'right': (0, 3), 'boat': 'right'}

Step 4 : {'left': (3, 1), 'right': (0, 2), 'boat': 'left'}

Step 5 : {'left': (1, 1), 'right': (2, 2), 'boat': 'right'}

Step 6 : {'left': (2, 2), 'right': (1, 1), 'boat': 'left'}

Step 7 : {'left': (0, 2), 'right': (3, 1), 'boat': 'right'}

Step 8 : {'left': (0, 3), 'right': (3, 0), 'boat': 'left'}

Step 9 : {'left': (0, 1), 'right': (3, 2), 'boat': 'right'}

Step 10 : {'left': (0, 2), 'right': (3, 1), 'boat': 'left'}

Step 11 : {'left': (0, 0), 'right': (3, 3), 'boat': 'right'}

**Explanation**

1. `from collections import deque`: This line imports the `deque` class from the `collections` module. `deque` is a double-ended queue data structure that supports adding and removing elements efficiently from both ends.

2. `initial\_state = {'left': (3, 3), 'right': (0, 0), 'boat': 'left'}`: This line defines the initial state of the problem. It represents the positions of missionaries and cannibals on both sides of the river, with the boat initially on the left side.

3. `goal\_state = {'left': (0, 0), 'right': (3, 3), 'boat': 'right'}`: This line defines the goal state of the problem. It represents the positions of missionaries and cannibals on both sides of the river, with everyone on the right side.

4. `def is\_valid(state): ...`: This block defines a function `is\_valid` that checks whether a given state is valid according to the rules of the problem. It ensures that the number of missionaries on either side of the river is not less than the number of cannibals, and that the number of missionaries and cannibals on each side and in the boat is within the allowed range (0 to 3).

5. `def generate\_next\_states(current\_state): ...`: This block defines a function `generate\_next\_states` that generates all possible next states from the current state. It considers all combinations of moving 1 or 2 missionaries or cannibals from one side to the other.

6. `def bfs(initial\_state, goal\_state): ...`: This block defines the breadth-first search (BFS) function. It takes the initial state and the goal state as input and searches for a solution using BFS.

7. `visited = set()`: This line initializes an empty set called `visited` to keep track of visited states during the BFS traversal.

8. `queue = deque([(initial\_state, [])])`: This line initializes a deque called `queue` with a tuple containing the initial state and an empty list representing the path taken to reach that state. The initial state is added to the queue.

9. `while queue: ...`: This block starts a loop that continues until the queue is empty.

10. `current\_state, path = queue.popleft()`: This line dequeues the leftmost element from the queue, which represents the current state and the path taken to reach that state.

11. `if current\_state == goal\_state: ...`: This line checks if the current state is equal to the goal state. If it is, the function returns the path to reach the goal state.

12. `if tuple(current\_state['left'] + current\_state['right'] + (current\_state['boat'],)) in visited: ...`: This line checks if the current state has been visited before. If it has, the loop continues to the next iteration.

13. `for next\_state in generate\_next\_states(current\_state): ...`: This line iterates over all possible next states generated from the current state.

14. `queue.append((next\_state, path + [next\_state]))`: This line appends the next state and the updated path (including the next state) to the queue for further exploration.

15. `return None`: If no solution is found after exploring all possible states, the function returns None.

16. `solution = bfs(initial\_state, goal\_state)`: This line calls the BFS function with the initial state and goal state to find the solution.

17. `if solution: ...`: This line checks if a solution was found. If it was, it prints the number of steps and each state in the solution path.

18. `else: ...`: If no solution was found, this line prints a message indicating that no solution was found.

This code efficiently searches for a solution to the missionaries and cannibals problem using the breadth-first search algorithm and prints the solution if found.