**Department of Computer Engineering**

**Academic Term II: 23-24**

**Class: B.E (Computer), Sem – VI Subject Name: Artificial Intelligence Student Name: Roll No: 9538**

| **Practical No:** | **4** |
| --- | --- |
| **Title:** | Use BFS problem solving method for  a) Water Jug Problem  b) Missionaries & Cannibals |
| **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(submitte  d) |  |
| **Total** | | | | |  |

**Signature of the Teacher:**

a) Missionaries & Cannibals

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:

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 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

b) Water Jug

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:

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