Name:Krish Gupta

Srn:202101521

AI Assignment 2

Implement the Water Jug problem using the following uninformed search strategies::

a. Depth First Search

Code:

class *State*:

    def \_\_init\_\_(self, jug1, jug2):

*self*.jug1 = jug1

*self*.jug2 = jug2

    def \_\_eq\_\_(self, other):

        return *self*.jug1 == other.jug1 and *self*.jug2 == other.jug2

    def \_\_hash\_\_(self):

        return hash((*self*.jug1, *self*.jug2))

    def \_\_str\_\_(self):

        return f"({*self*.jug1}, {*self*.jug2})"

class *WaterJugProblem*:

    def \_\_init\_\_(self, capacity\_jug1, capacity\_jug2, target):

*self*.capacity\_jug1 = capacity\_jug1

*self*.capacity\_jug2 = capacity\_jug2

*self*.target = target

    def is\_valid\_state(self, state):

        return 0 <= state.jug1 <= *self*.capacity\_jug1 and 0 <= state.jug2 <= *self*.capacity\_jug2

    def dfs(self, current\_state, visited):

        if current\_state == *self*.target:

            return [current\_state]

        visited.add(current\_state)

        next\_states = [

            State(0, current\_state.jug2),  *# Empty jug1*

            State(current\_state.jug1, 0),  *# Empty jug2*

            State(*self*.capacity\_jug1, current\_state.jug2),  *# Fill jug1*

            State(current\_state.jug1, *self*.capacity\_jug2),  *# Fill jug2*

            State(min(*self*.capacity\_jug1, current\_state.jug1 + current\_state.jug2),

                  max(0, current\_state.jug1 + current\_state.jug2 - *self*.capacity\_jug1)),  *# Pour from jug2 to jug1*

            State(max(0, current\_state.jug1 + current\_state.jug2 - *self*.capacity\_jug2),

                  min(*self*.capacity\_jug2, current\_state.jug1 + current\_state.jug2))  *# Pour from jug1 to jug2*

        ]

        for next\_state in next\_states:

            if next\_state not in visited and *self*.is\_valid\_state(next\_state):

                path = *self*.dfs(next\_state, visited)

                if path:

                    return [current\_state] + path

        return None

    def solve(self):

        initial\_state = State(0, 0)

        visited = set()

        path = *self*.dfs(initial\_state, visited)

        if path:

            return path

        else:

            return "No solution found."

*# Example usage:*

capacity\_jug1 = 4

capacity\_jug2 = 3

target = State(2, 0)

problem = WaterJugProblem(capacity\_jug1, capacity\_jug2, target)

solution = problem.solve()

print("Solution:")

for state in solution:

    print(state)

Output:

Solution:

(0, 0)

(4, 0)

(4, 3)

(0, 3)

(3, 0)

(3, 3)

(4, 2)

(0, 2)

(2, 0)

b. Breadth First Search

Code:

from collections import deque

class *State*:

    def \_\_init\_\_(self, jug1, jug2):

*self*.jug1 = jug1

*self*.jug2 = jug2

    def \_\_eq\_\_(self, other):

        return *self*.jug1 == other.jug1 and *self*.jug2 == other.jug2

    def \_\_hash\_\_(self):

        return hash((*self*.jug1, *self*.jug2))

    def \_\_str\_\_(self):

        return f"({*self*.jug1}, {*self*.jug2})"

class *WaterJugProblem*:

    def \_\_init\_\_(self, capacity\_jug1, capacity\_jug2, target):

*self*.capacity\_jug1 = capacity\_jug1

*self*.capacity\_jug2 = capacity\_jug2

*self*.target = target

    def is\_valid\_state(self, state):

        return 0 <= state.jug1 <= *self*.capacity\_jug1 and 0 <= state.jug2 <= *self*.capacity\_jug2

    def bfs(self):

        initial\_state = State(0, 0)

        visited = set()

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

        while queue:

            current\_state, path = queue.popleft()

            if current\_state == *self*.target:

                return path + [current\_state]

            visited.add(current\_state)

            next\_states = [

                State(0, current\_state.jug2),  *# Empty jug1*

                State(current\_state.jug1, 0),  *# Empty jug2*

                State(*self*.capacity\_jug1, current\_state.jug2),  *# Fill jug1*

                State(current\_state.jug1, *self*.capacity\_jug2),  *# Fill jug2*

                State(min(*self*.capacity\_jug1, current\_state.jug1 + current\_state.jug2),

                      max(0, current\_state.jug1 + current\_state.jug2 - *self*.capacity\_jug1)),  *# Pour from jug2 to jug1*

                State(max(0, current\_state.jug1 + current\_state.jug2 - *self*.capacity\_jug2),

                      min(*self*.capacity\_jug2, current\_state.jug1 + current\_state.jug2))  *# Pour from jug1 to jug2*

            ]

            for next\_state in next\_states:

                if next\_state not in visited and *self*.is\_valid\_state(next\_state):

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

        return "No solution found."

*# Example usage:*

capacity\_jug1 = 4

capacity\_jug2 = 3

target = State(2, 0)

problem = WaterJugProblem(capacity\_jug1, capacity\_jug2, target)

solution = problem.bfs()

print("Solution:")

for state in solution:

    print(state)

Output:

Solution:

(0, 0)

(0, 3)

(3, 0)

(3, 3)

(4, 2)

(0, 2)

(2, 0)