Water Jug Problem in AI: The Complete Guide

The water Jug Problem, as the name suggests, is a problem where two jugs of water are given, say one is a 4-litre one, and the other one is a 3-litre one, but none of the measuring markers is mentioned on any of it. There is a pump available to fill the jugs with water. How can you exactly pour 2 litres of water into a 4-litre jug? Assuming that both the jugs are empty, the task is to find a solution to pour 2-litre water into a 4-litre jug.

**Production Rules for the Water Jug Problem in Artificial Intelligence**

To solve the water jug problem, many algorithms can be used. These include:

* Breadth-First Search: [BFS or Breadth First Search](https://www.simplilearn.com/tutorials/data-structure-tutorial/bfs-algorithm) visits the nodes in order of their distance from the starting node. This implies that it will visit the nearest node first.
* Depth First Search: [DFS or Depth First Search](https://www.simplilearn.com/tutorials/data-structure-tutorial/dfs-algorithm) visits the nodes in order of their depth.

In production rules for the water jug problem, let x denote a 4-litre jug, and y denote a 3-litre jug, i.e. x=0,1,2,3,4 or y=0,1,2,3

**1.**[**Breadth-First Search (BFS)**](https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/)

* BFS explores all possible states level by level, ensuring that the shortest path (fewest operations) is found. It is particularly useful for the Water Jug Problem as it guarantees finding the optimal solution.
* BFS starts from the initial state **(0, 0)** and explores all neighboring states, then their neighbors, and so on until the goal state is found.

**2.**[**Depth-First Search (DFS)**](https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/)

* DFS explores each path from the initial state as deeply as possible before backtracking. While DFS can find a solution, it does not guarantee the optimal one and may result in exploring longer paths unnecessarily.
* DFS works well for smaller problems but may struggle with larger state spaces due to its depth-first nature.

**The Solution to the Water Jug Problem in Artificial Intelligence**

Here is the water jug problem solution in AI is as follows:

Current state (0,0)

Loop till the goal state (2,0) is reached.

* Apply a rule when the left side matches the current state
* Set the new current state to the resulting state

Start state (0,0)

(0,3) Apply Rule 2, Fill the 3-litre Jug

(3,0) Apply Rule 9: Pour all the water from a 3-litre jug into a 4-litre jug

(3,3) Apply Rule 2, Fill the 3-litre Jug

(4,2) Apply Rule 7: Pour water from a 3-litre jug into a 4-litre jug until it is full

(0,2) Apply Rule 5, Empty 4-litre jug on the ground

(2,0) Apply Rule 9: Pour all the water from a 3-litre jug into a 4-litre jug

Another water jug problem solution is:

(0, 0) – Start State

(4, 0) – Rule 1: Fill the 4-litre jug

(1, 3) – Rule 8: Pour water from the 4-litre jug into the 3-litre jug until the 3-litre jug is full.

(1, 0) – Rule 6: Empty the 3-litre jug on the ground

(0, 1) – Rule 10: Pour all the water from the 4-litre jug into the 3-litre jug.

(4, 1) –  Rule 1: Fill the 4-litre jug

.(2, 3) – Rule 8: Pour water from the 4-litre jug into the 3-litre jug until the 3-litre jug is full.

Goal State reached

**1. What is a water jug problem in AI?**

The water jug problem in AI is a complex problem where it is important to find a way to measure the amount of water using two jugs of different capacities.

**2. How do you solve the water jug problem in AI?**

To solve the water jug problem in AI, Breadth First Search is the best water jug problem solution.

**3. Which algorithm is used for water jug problems?**

Algorithms like Breadth-First Search, Depth-First Search, Heuristic Search, and Stace-Space Representation can be used to solve the water jug problem.

**Python Implementation: Solving Water Jug Problem Using Depth First Search**

# Function to perform DFS to solve the water jug problem

def water\_jug\_dfs(capacity1, capacity2, target):

visited = set() # To track visited states

path = [] # To store the solution path

def dfs(jug1, jug2):

# If we have already visited this state, return False (avoid cycles)

if (jug1, jug2) in visited:

return False

# Mark the state as visited

visited.add((jug1, jug2))

# Append the current state to the path

path.append((jug1, jug2))

# If the target is achieved in either jug, return True

if jug1 == target or jug2 == target:

return True

# Explore all possible transitions (DFS recursive calls)

# Fill 3-liter jug

if dfs(3, jug2):

return True

# Fill 5-liter jug

if dfs(jug1, 5):

return True

# Empty 3-liter jug

if dfs(0, jug2):

return True

# Empty 5-liter jug

if dfs(jug1, 0):

return True

# Pour water from 3-liter jug into 5-liter jug

if dfs(max(0, jug1 - (5 - jug2)), min(5, jug1 + jug2)):

return True

# Pour water from 5-liter jug into 3-liter jug

if dfs(min(3, jug1 + jug2), max(0, jug2 - (3 - jug1))):

return True

# If none of the transitions lead to the goal, backtrack

path.pop()

return False

# Start DFS from the initial state (0, 0)

dfs(0, 0)

# If we found a solution, return the path

return path

# Example Usage

capacity1 = 3 # Capacity of the 3-liter jug

capacity2 = 5 # Capacity of the 5-liter jug

target = 4 # Target amount to measure

solution = water\_jug\_dfs(capacity1, capacity2, target)

if solution:

print("Solution steps:")

for step in solution:

print(step)

else:

print("No solution found.")

import matplotlib.pyplot as plt

import networkx as nx

# Function to create and visualize the state space transitions for DFS

def visualize\_dfs\_solution(solution):

G = nx.DiGraph()

# Add the nodes and edges based on the DFS solution path

for i in range(len(solution) - 1):

G.add\_edge(solution[i], solution[i + 1])

pos = nx.spring\_layout(G) # Position the nodes for visualization

plt.figure(figsize=(8, 6))

# Draw the graph with nodes and labels

nx.draw(G, pos, with\_labels=True, node\_color='lightgreen', node\_size=1500, font\_size=12, font\_weight='bold')

nx.draw\_networkx\_edges(G, pos, edgelist=list(G.edges()), edge\_color='blue', width=2)

plt.title("Water Jug Problem - DFS Solution Path")

plt.show()

# Visualize the DFS solution

if solution:

visualize\_dfs\_solution(solution)

**Python Implementation: Solving Water Jug Problem Using Breath First Search**

**Algorithm-**

* Initialise 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:
  + Pop out the first element of the queue.
  + If the value of popped element is equal to **Z**, return True.
  + Let **X\_left**and **Y\_left** be the amount of water left in the jugs respectively.
  + Now perform the **fill** operation:
    - 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.
    - 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.
  + Perform the **empty** operation:
    - If the state **({0, Y\_left})** isn’t visited, insert it into the hashmap, since we can empty any of the jugs.
    - Similarly, if the state **({X\_left, 0)** isn’t visited, insert it into the hashmap, since we can empty any of the jugs.
  + Perform the **transfer of water** operation:
    - **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 hashmap.
    - **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 hashmap.
* Return False, since, it is not possible to measure **Z** litres.

**Source code -**

from collections import deque

# Function to find the minimum operations to obtain

# d liters in one jug

def min\_steps(m, n, d):

if d > max(m, n):

return -1

# Queue for BFS: (jug1, jug2, steps)

q = deque([(0, 0, 0)])

# For tracking the visited states

visited = [[False] \* (n + 1) for \_ in range(m + 1)]

visited[0][0] = True

while q:

jug1, jug2, steps = q.popleft()

if jug1 == d or jug2 == d:

return steps

# 1: Fill jug1

if not visited[m][jug2]:

visited[m][jug2] = True

q.append((m, jug2, steps + 1))

# 2: Fill jug2

if not visited[jug1][n]:

visited[jug1][n] = True

q.append((jug1, n, steps + 1))

# 3: Empty jug1

if not visited[0][jug2]:

visited[0][jug2] = True

q.append((0, jug2, steps + 1))

# 4: Empty jug2

if not visited[jug1][0]:

visited[jug1][0] = True

q.append((jug1, 0, steps + 1))

# 5: Pour jug1 into jug2

pour1to2 = min(jug1, n - jug2)

if not visited[jug1 - pour1to2][jug2 + pour1to2]:

visited[jug1 - pour1to2][jug2 + pour1to2] = True

q.append((jug1 - pour1to2, jug2 + pour1to2, steps + 1))

# 6: Pour jug2 into jug1

pour2to1 = min(jug2, m - jug1)

if not visited[jug1 + pour2to1][jug2 - pour2to1]:

visited[jug1 + pour2to1][jug2 - pour2to1] = True

q.append((jug1 + pour2to1, jug2 - pour2to1, steps + 1))

return -1

if \_\_name\_\_ == "\_\_main\_\_":

# jug1 = 4 litre, jug2 = 3 litre

m, n, d = 4, 3, 2

print(min\_steps(m, n, d))