**Mahavir Education Trust's**

**Shah & Anchor Kutchhi Engineering College,**

**Chembur, Mumbai 400 088**

**UG Program in Information Technology**

**Name:** GARIMA MAHENDRA MAHTO

**Class-Batch:** BE\_5A

**Roll No:** 07

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **EXPERIMENT\_NO.05** | | | | | |
| **Date of Performance:** | 09/09/2021 | | | | |
| **Date of Submission:** | 16/09/2021 | | | | |
| **Program formation/ Execution/ ethical practices (07)** | **Documentation (02)** | **Timely Submission (03)** | **Viva Answer**  **(03)** | **Experiment Marks**  **(15)** | **Teacher Signature**  **with date** |
|  |  |  |  |  |  |

**EXPERIMENT\_NO.05**

**AIM:** Uninformed search techniques:(Any one)

●Implement pathfinding in maze using depth-first search (DFS)

●Implement water jug problem using breadth-first search (BFS).

**LAB OUTCOME:** 7. ITL703.2 Analyse and formalize the problem as a state space, graph, design heuristics and select amongst different search or game-based techniques to solve them.

**THEORY:**

**Breadth-first search** (**BFS**) is an [algorithm](https://en.wikipedia.org/wiki/Algorithm) for searching a [tree](https://en.wikipedia.org/wiki/Tree_(data_structure)) data structure for a node that satisfies a given property. It starts at the [tree root](https://en.wikipedia.org/wiki/Tree_(data_structure)#Terminology) and explores all nodes at the present [depth](https://en.wikipedia.org/wiki/Tree_(data_structure)#Terminology) prior to moving on to the nodes at the next depth level. Extra memory, usually a [queue](https://en.wikipedia.org/wiki/Queue_(data_structure)), is needed to keep track of the child nodes that were encountered but not yet explored.

**Depth-first search** (**DFS**) is an [algorithm](https://en.wikipedia.org/wiki/Algorithm) for traversing or searching [tree](https://en.wikipedia.org/wiki/Tree_data_structure) or [graph](https://en.wikipedia.org/wiki/Graph_(data_structure)) data structures. The algorithm starts at the [root node](https://en.wikipedia.org/wiki/Tree_(data_structure)#Terminology) (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

**Water Jug problem using BFS**

You are given a m litter jug and a n litter 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 litters 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:

Empty a Jug, (X, Y)->(0, Y) Empty Jug 1

Fill a Jug, (0, 0)->(X, 0) Fill Jug 1

Pour water from one jug to the other until one of the jugs is either empty or full, (X, Y) -> (X-d, Y+d)

Example: Input: 4 3 2

Output: {(0, 0), (0, 3), (4, 0), (4, 3),

(3, 0), (1, 3), (3, 3), (4, 2),

(0, 2)}

**Code:**

from collections import deque

def BFS (a, b, target):

# Map is used to store the states, every

# State is hashed to binary value to

# Indicate either that state is visited

# Before or not

m = {}

isSolvable = False

path = []

# Queue to maintain states

q = deque()

# Initialing with initial state

q.append((0, 0))

while (len(q) > 0):

# Current state

u = q.popleft()

#q.pop() #pop off used state

# If this state is already visited

if ((u[0], u[1]) in m):

continue

# Doesn't met jug constraints

if ((u[0] > a or u[1] > b or

u[0] < 0 or u[1] < 0)):

continue

# Filling the vector for constructing

# the solution path

path.append([u[0], u[1]])

# Marking current state as visited

m[(u[0], u[1])] = 1

# If we reach solution state, put ans=1

if (u[0] == target or u[1] == target):

isSolvable = True

if (u[0] == target):

if (u[1] != 0):

# Fill final state

path.append([u[0], 0])

else:

if (u[0] != 0):

# Fill final state

path.append([0, u[1]])

# Print the solution path

sz = len(path)

for i in range(sz):

print("(", path[i][0], ",",

path[i][1], ")")

break

# If we have not reached final state

# then, start developing intermediate

# states to reach solution state

q.append([u[0], b]) # Fill Jug2

q.append([a, u[1]]) # Fill Jug1

for ap in range(max(a, b) + 1):

# Pour amount ap from Jug2 to Jug1

c = u[0] + ap

d = u[1] - ap

# Check if this state is possible or not

if (c == a or (d == 0 and d >= 0)):

q.append([c, d])

# Pour amount ap from Jug 1 to Jug2

c = u[0] - ap

d = u[1] + ap

# Check if this state is possible or not

if ((c == 0 and c >= 0) or d == b):

q.append([c, d])

# Empty Jug2

q.append([a, 0])

# Empty Jug1

q.append([0, b])

# No, solution exists if ans=0

if (not isSolvable):

print ("No solution")

# Driver code

if \_\_name\_\_ == '\_\_main\_\_':

Jug1, Jug2, target = 4, 3, 2

print("Path from initial state "

"To solution state: “)

BFS (Jug1, Jug2, target)

**Output:** Path from initial state to solution state:

0 0

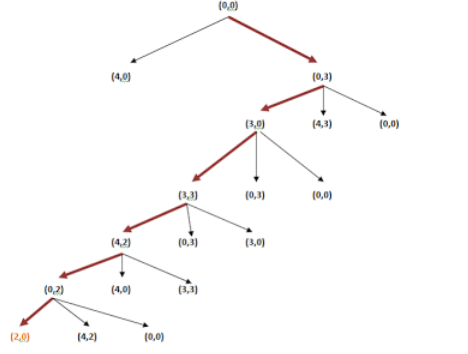
5 0

0 5

5 5

3 7

3 0

**Tree represent:** 

**CONCLUSION:** The BFS approach for solving the water jug problem has been one of the most feasible solutions. Although, various programmers have solved the Water Jug problem using recursion and other tree exploitation strategies. This can be written in any of the programming languages including development-based languages such as Python, JavaScript, Swift, etc.

Algorithms are updated very swiftly, still, Artificial Intelligence experts are working to find a better solution for solving the Water Jug problem. Analytically, a better solution would be a hybrid algorithm, devised by combining the properties of breadth-first search, depth-first search, recursive function calls, priority approach.