SHRIA SRIVASTAVA
2019140064
TE IT
BATCH D

Problem Statement:

IMPLEMENT WATER JUG PROBLEM USING UNINFORMED SEARCH STRATEGY.

a. Formulate the problem precisely, making only those distinctions necessary to ensure a valid solution. Draw a diagram of the PARTIAL state space.

b. Implement and solve the problem using an uninformed search strategy. BFS and DFS

c. Compare both search strategies against evaluation criteria

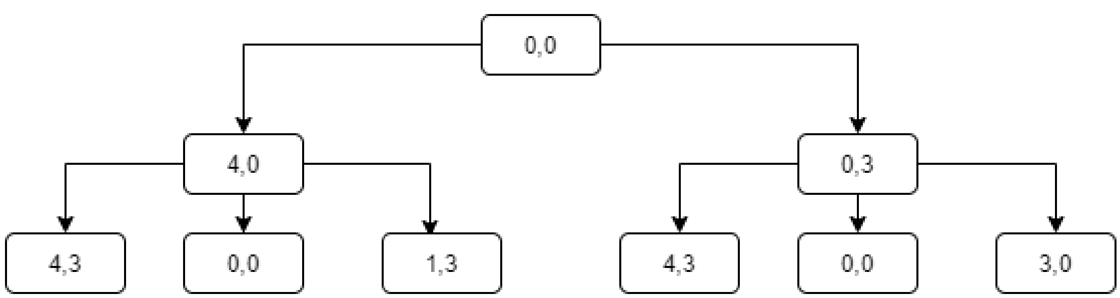
1-Completeness

2- Optimal

3- Time complexity

4-Space Complexity

State Space Tree:



Performance measure:

If the target jug (x) is filled with target amount of water (2 gallons) or not

Environment of Agent:

The jugs and their water levels is the environment for the agent

Actuators:

The actions that can be taken according to the production rules are the actuators.

Sensors:

The program itself that keeps the track of the water levels in the two jugs is the sensor.

Code:

BFS:

```
            if psn.x == position.x and psn.y ==
position.y: #Check if visited
                return 0
        queue.append(position)
        count +=1
        return 0
    else:
        for psn in queue:
            if psn.x == position.x and psn.y ==
position.y:
                return 1
        queue.append(position)
        count +=1
        return 1

#j1, j2 hold max jug capacities
# jt1 stores the target level for jug 1
def BFS(j1, j2, jt1, queue):
    while 1:
        #print(queue)
        node=queue.pop(0)
        i = node.x
        j = node.y
        can_fill1 = j1 - i  #remaining capacities of
jugs
        can_fill2 = j2 - j

        # Production rules
        if can_fill1 > j:
            a = position(i+j, 0, node)
            if push_position(queue, a, jt1):
                break
```

```
        else :
            b = position(j1, j - can_fill1, node)
            if push_position(queue, b, jt1):
                break
        if can_fill2 > i:
            c = position(0, i+j, node)
            if push_position(queue, c, jt1):
                break
        else :
            d = position(i - can_fill2, j2, node)
            if push_position(queue, d, jt1):
                break
        e = position(j1, j, node)
        f = position(i, j2, node)
        g = position(0, j, node)
        h = position(i, 0, node)
        if push_position(queue, e, jt1):
                break
        if push_position(queue, f, jt1):
                break
        if push_position(queue, g, jt1):
                break
        if push_position(queue, h, jt1):
                break




#Driver Code
jug1_capacity = 4
jug2_capacity = 3
initial_state = position(0, 0, None)
```

```
jug1_target = 2
queue = [initial_state]
BFS(jug1_capacity, jug2_capacity, jug1_target, queue)

temp = queue[-1]
array = []
# Reverse tracing from solution to root
while temp.parent != None:
    array.append((temp.x, temp.y))
    temp = temp.parent
array.append((temp.x, temp.y))

array.reverse()
for elementin array:
    print(element)

print(f"No of nodes generated is {count}")
```

DFS:

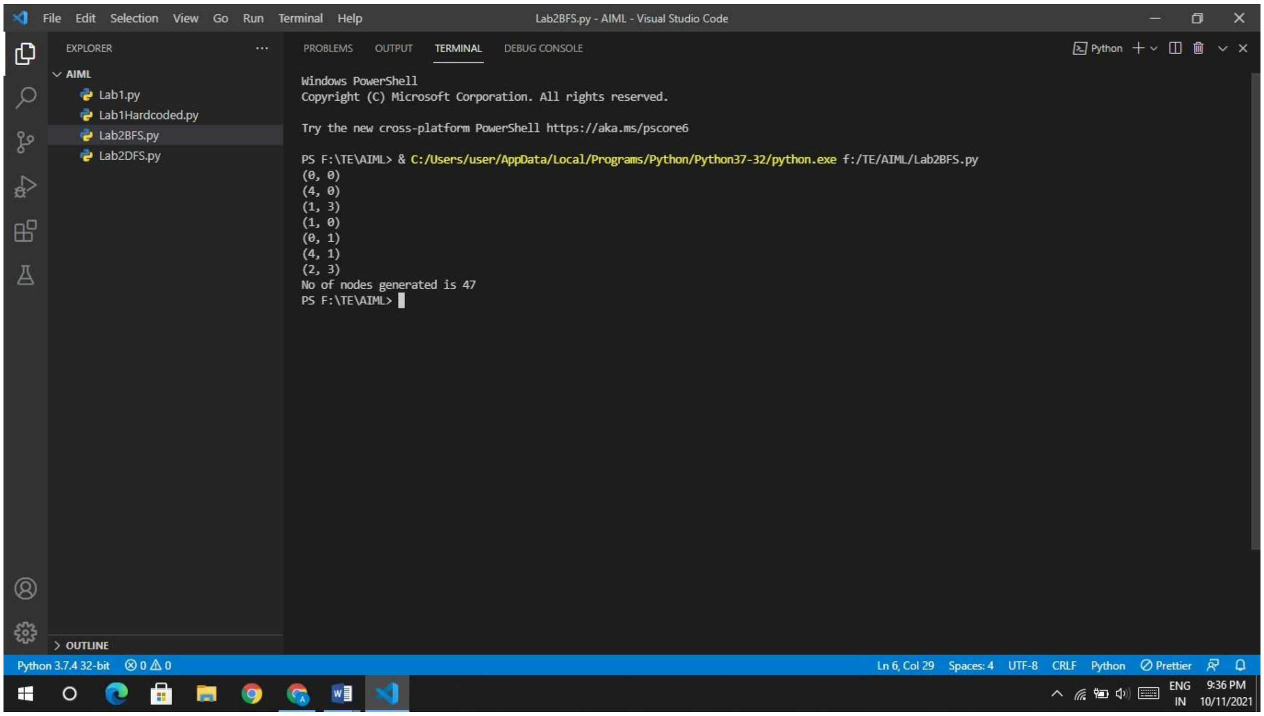
```
found = 0             #Global variable to check if
solution is found
no_of_nodes = 0       #Global variable to count no of
nodes


#prev_states holds the visited nodes
#start1, start2 hold current jug levels
#j1, j2 hold max jug capacities
# jt1 stores the target level for jug 1
def DFS(start1, start2, j1, j2, jt1, prev_states):
```

```
    global no_of_nodes
    #Check for invalid conditions
    if start1 >=0 and start2 >=0 and start1 <= j1 and
start2 <=j2:

        if (start1, start2) not in
prev_states:#Repitition check
            no_of_nodes += 1
            global found
            if found == 0: # If solution is not found
                prev_states.append((start1, start2))
                if start1 == jt1:
                    found = 1
                    return
                can_fill1 = j1 - start1 #remaining
capacities of jugs
                can_fill2 = j2 - start2


                # Production rules
                if can_fill1 > start2:
                    DFS(start1 + start2, 0, j1, j2, jt1,
prev_states)
                else:
                    DFS(start1 + can_fill1, start2 -
can_fill1, j1, j2, jt1, prev_states)
                if can_fill2 > start1:
                    DFS(0, start2 + start1, j1, j2, jt1,
prev_states)
                else:
                    DFS(start1 - can_fill2, start2 +
can_fill2, j1, j2, jt1, prev_states)
```

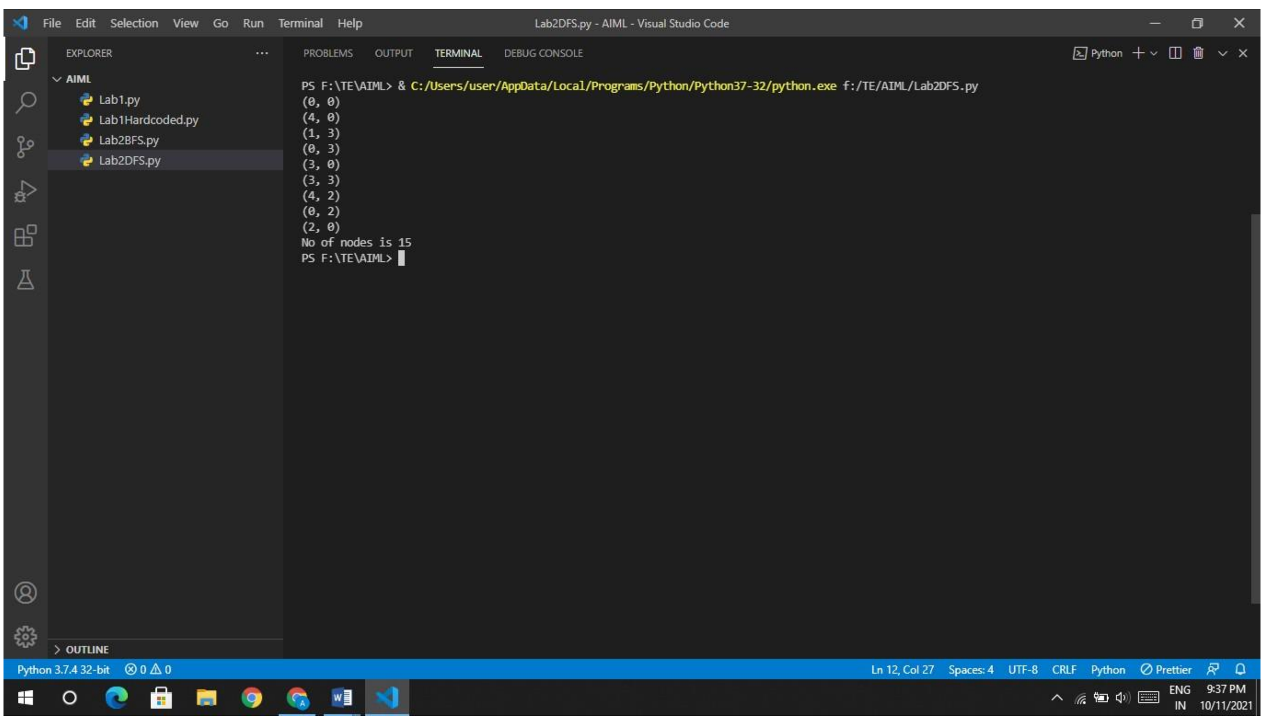
```
                DFS(0, start2, j1, j2, jt1, prev_states)
                DFS(start1, 0, j1, j2, jt1, prev_states)
                DFS(j1, start2, j1, j2, jt1,
prev_states)
                DFS(start1, j2, j1, j2, jt1,
prev_states)



#Driver Code
jug1_capacity = 4
jug2_capacity  = 3
jug1_target = 2
prev_states = []
DFS(0, 0, jug1_capacity, jug2_capacity, jug1_target,
prev_states)
for state in prev_states:
    print(state)
print(f"No of nodes is {no_of_nodes}")
```

Output:

DFS:



BFS:



Observation:

In both DFS and BFS, the order in which production rules are executed changes the number of nodes generated.

Conclusion:

In the water jug problem, the state space tree is infinite because of loops (filling and emptying) thus it useful to do a graph search and maintain the visited nodes to avoid repetition.

In general BFS generates more nodes but gives shallowest (most efficient solution) while DFS creates less nodes but does not necessarily give the shallowest solution (or any solution at all).

Comparison:

	BFS	**DFS**
COMPLETE	yes	no
OPTIMAL	yes	no
TIME COMPLEXITY	Depends on branching factor	Depends on height of state space tree
SPACE COMPLEXITY	Greater than DFS	Lesser than BFS