**Name: Alekhya Tejomurtula**

**Class: TY A**

**Roll No: 58**

**SRN: 201900449**

**AI ASSIGNMENT-2**

**A) Water jug problem using Depth First Search**

**CODE:**

def LevelOrderTraversal(root):

if (root == None):

return;

# Standard level order traversal code

# using queue

q = [] # Create a queue

q.append(root); # Enqueue root

while (len(q) != 0):

n = len(q);

# If this node has children

while (n > 0):

# Dequeue an item from queue and print it

p = q[0]

q.pop(0)

print(p.list, end=' ')

# Enqueue all children of the dequeued item

for i in range(len(p.child)):

q.append(p.child[i]);

n -= 1

print() # Print new line between two levels

print("-------")

def fill4LitreJug(list):

if list[0] == 4:

return False

list[0] = 4

return list

def fill3LitreJug(list):

if list[1] == 3:

return False

list[1] = 3

return list

def empty3(list):

listt = list.copy()

if listt[1] == 0:

return False

listt[1] = 0

return listt

def empty4(list):

listt = list.copy()

if listt[0] == 0:

return False

listt[0] = 0

return listt

def transferFrom\_3to4(listt):

if listt[0]==4:

return False

elif listt[1] == 0:

return False

elif listt[0] < 4:

if (4 - listt[0]) >= listt[1]:

listt[0] = listt[0] + listt[1]

listt[1] = 0

else:

emptySpace = 4 - listt[0]

listt[1] = listt[1] - emptySpace

listt[0] = listt[0] + emptySpace

return list

def transferFrom\_4to3(listt):

if listt[1]==3:

return False

elif listt[0] == 0:

return False

elif listt[1] < 3:

if (3 - listt[1]) >= listt[0]:

listt[1] = listt[1] + listt[0]

listt[0] = 0

else:

emptySpace = 3 - listt[1]

listt[0] = listt[0] - emptySpace

listt[1] = listt[1] + emptySpace

return listt

GOAL = [2,0]

class Node:

def \_\_init\_\_(self, list):

self.list = list

self.child = []

self.myParents = []

# Utility function to create a new tree node

def newNode(key):

temp = Node(key)

return temp

# Prints the n-ary tree level wise

answerslist = []

#list = [0 , 0]

#-------------------------------------------------------------------

def findOptimalPath(node):

if node.myParents:

if node.myParents[-1] ==GOAL:

#print("--",node.myParents[-1],"--")

answerslist.append(node.myParents)

if node.list in node.myParents:

return

childrenlist = []

#print("Passed Node: ",node.list)

list1 = empty4((node.list).copy())

list2 = empty3((node.list).copy())

list3 = transferFrom\_3to4((node.list).copy())

list4 = transferFrom\_4to3((node.list).copy())

list5 = fill4LitreJug((node.list).copy())

list6 = fill3LitreJug((node.list).copy())

#print("lists: ", list1, list2 ,list3, list4 , list5 , list6)

childrenlist.extend((list1, list2, list3,list4,list5,list6))

childrenlist = [x for x in childrenlist if x is not False]

#print("Childrenlist: ",childrenlist)

#print("\n")

for i in range(0,len(childrenlist)):

(node.child).append(newNode(childrenlist[i]))

node.child[i].myParents.extend(node.myParents)

node.child[i].myParents.append(node.list)

if node.child[i].myParents[-1] ==GOAL:

answerslist.append(node.child[i].myParents)

return

#print("myParents: ", i +1," : ", node.child[i].myParents )

findOptimalPath(node.child[i])

return

mainlist= [0, 0]

root = newNode(mainlist)

findOptimalPath(root)

#print("AnswersList: ", answerslist)

print("AnswersList")

print('\n'.join(map(str, answerslist)))

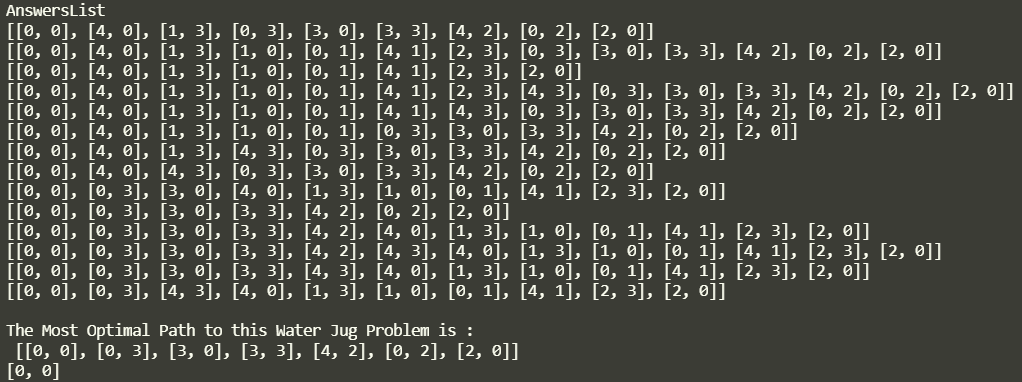
smallest = []

for i in answerslist:

smallest.append(len(i))

print("\nThe Most Optimal Path to this Water Jug Problem is :\n", answerslist[smallest.index(min(smallest))])

LevelOrderTraversal(root)

**OUTPUT:**

**B) Water jug problem using Breadth First Search**

**CODE:**

LEFT\_BUCKET\_CAPACITY = 4

RIGHT\_BUCKET\_CAPACITY =3

GOAL = (0, 2)

RESULT = []

def move\_left\_to\_right(jug):

allowed\_space = min(RIGHT\_BUCKET\_CAPACITY - jug[1], jug[0])

return (jug[0] - allowed\_space, jug[1] + allowed\_space)

def move\_right\_to\_left(jug):

allowed\_space = min(LEFT\_BUCKET\_CAPACITY - jug[0], jug[1])

return (jug[0] + allowed\_space, jug[1] - allowed\_space)

def empty\_left(jug):

return (0, jug[1])

def empty\_right(jug):

return (jug[0], 0)

def fill\_left(jug):

return (LEFT\_BUCKET\_CAPACITY, jug[1])

def fill\_right(jug):

return (jug[0], RIGHT\_BUCKET\_CAPACITY)

def get\_available\_operations(jug):

operations = {

move\_left\_to\_right,

move\_right\_to\_left,

empty\_left,

empty\_right,

fill\_left,

fill\_right

}

# if left jug is empty

if jug[0] == 0:

operations.remove(empty\_left)

operations.remove(move\_left\_to\_right)

# if left jug is full

elif jug[0] == LEFT\_BUCKET\_CAPACITY:

operations.remove(fill\_left)

operations.remove(move\_right\_to\_left)

# if right jug is empty

if jug[1] == 0:

operations.remove(empty\_right)

try: operations.remove(move\_right\_to\_left)

except KeyError: pass

# if right jug is full

elif jug[1] == RIGHT\_BUCKET\_CAPACITY:

operations.remove(fill\_right)

try: operations.remove(move\_left\_to\_right)

except KeyError: pass

return operations

def get\_operation\_name(operation) -> str:

return {

fill\_left: 'fill left jug',

fill\_right: 'fill right jug',

empty\_left: 'empty left jug',

empty\_right: 'empty right jug',

move\_left\_to\_right: 'pour left jug into right jug',

move\_right\_to\_left: 'pour right jug into left jug',

}[operation]

class Node:

def \_\_init\_\_(self, jug: tuple[int, int], parent = None, operation\_name: str = None) -> None:

self.jug = jug

self.parent = parent

self.operation\_name = operation\_name

def grow\_tree(parent: Node, previous = {(0, 0)}) -> bool:

queue = [parent]

opened = []

closed = []

level = 1

while len(queue) != 0:

node = queue.pop(0) # Remove first elemnt from queue

opened.append(node.jug)

operations = get\_available\_operations(node.jug)

# Iterate over all operations for current node

# Assign the child nodes to parent

for op in operations:

child\_jug = op(node.jug)

child = Node(child\_jug, node, get\_operation\_name(op))

closed.append(child\_jug)

if child\_jug == GOAL:

RESULT.append(child)

return True

if child\_jug in previous:

continue

else:

previous.add(child\_jug)

queue.append(child)

print(f" At breadth level {level} ".center(40, '='))

print("Opened list: ", opened)

print("Closed list: ", closed, end='\n\n')

level += 1

return False

def main():

seed = Node((0, 0))

if grow\_tree(seed):

print("=" \* 40)

print("The full path is")

for endpoint in RESULT:

path = []

operations: list[str] = []

while endpoint.parent:

path.append(endpoint.jug)

operations.append(endpoint.operation\_name)

endpoint = endpoint.parent

path = list(reversed(path))

operations = list(reversed(operations))

print("From (0, 0)")

for i, \_ in enumerate(path):

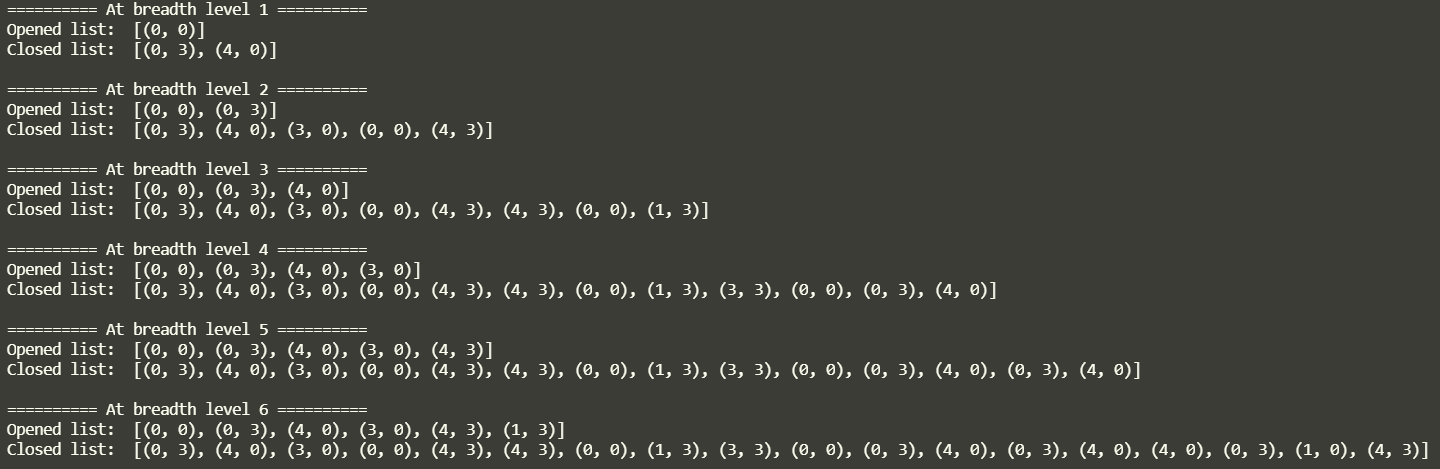
print(f'Step {i + 1} {operations[i].ljust(30)} => {path[i]}')

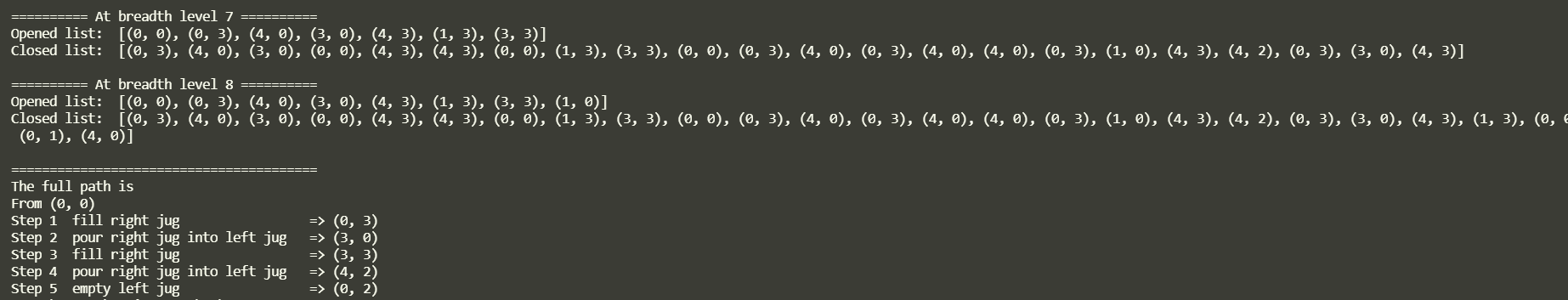
else:

print("Could not reach the goal", GOAL)

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

main()

**OUTPUT:**

****