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
```

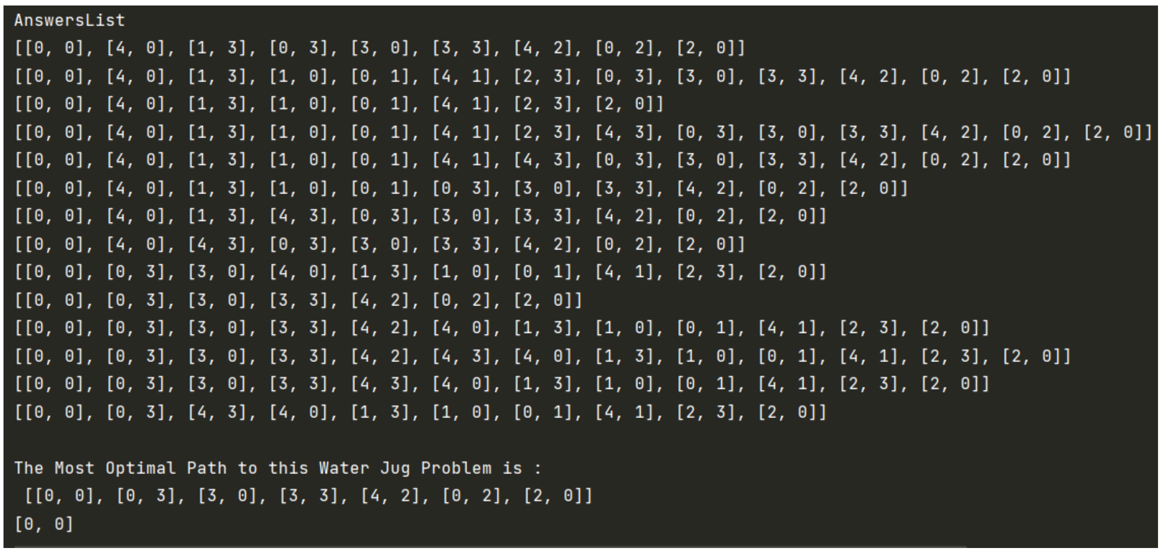
```
        print() # Print new line between two levels
        print("-------")

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:



```
AnswersList
[[0, 0], [4, 0], [1, 3], [0, 3], [3, 0], [3, 3], [4, 2], [0, 2], [2, 0]]
[[0, 0], [4, 0], [1, 3], [1, 0], [0, 1], [4, 1], [2, 3], [0, 3], [3, 0], [3, 3], [4, 2], [0, 2], [2, 0]]
[[0, 0], [4, 0], [1, 3], [1, 0], [0, 1], [4, 1], [2, 3], [2, 0]]
[[0, 0], [4, 0], [1, 3], [1, 0], [0, 1], [4, 1], [2, 3], [4, 3], [0, 3], [3, 0], [3, 3], [4, 2], [0, 2], [2, 0]]
[[0, 0], [4, 0], [1, 3], [1, 0], [0, 1], [4, 1], [4, 3], [0, 3], [3, 0], [3, 3], [4, 2], [0, 2], [2, 0]]
[[0, 0], [4, 0], [1, 3], [1, 0], [0, 1], [0, 3], [3, 0], [3, 3], [4, 2], [0, 2], [2, 0]]
[[0, 0], [4, 0], [1, 3], [4, 3], [0, 3], [3, 0], [3, 3], [4, 2], [0, 2], [2, 0]]
[[0, 0], [4, 0], [4, 3], [0, 3], [3, 0], [3, 3], [4, 2], [0, 2], [2, 0]]
[[0, 0], [0, 3], [3, 0], [4, 0], [1, 3], [1, 0], [0, 1], [4, 1], [2, 3], [2, 0]]
[[0, 0], [0, 3], [3, 0], [3, 3], [4, 2], [0, 2], [2, 0]]
[[0, 0], [0, 3], [3, 0], [3, 3], [4, 2], [4, 0], [1, 3], [1, 0], [0, 1], [4, 1], [2, 3], [2, 0]]
[[0, 0], [0, 3], [3, 0], [3, 3], [4, 2], [4, 3], [4, 0], [1, 3], [1, 0], [0, 1], [4, 1], [2, 3], [2, 0]]
[[0, 0], [0, 3], [3, 0], [3, 3], [4, 3], [4, 0], [1, 3], [1, 0], [0, 1], [4, 1], [2, 3], [2, 0]]
[[0, 0], [0, 3], [4, 3], [4, 0], [1, 3], [1, 0], [0, 1], [4, 1], [2, 3], [2, 0]]

The Most Optimal Path to this Water Jug Problem is :
 [[0, 0], [0, 3], [3, 0], [3, 3], [4, 2], [0, 2], [2, 0]]
[0, 0]
```

B) Water jug problem using Breadth First Search

CODE:

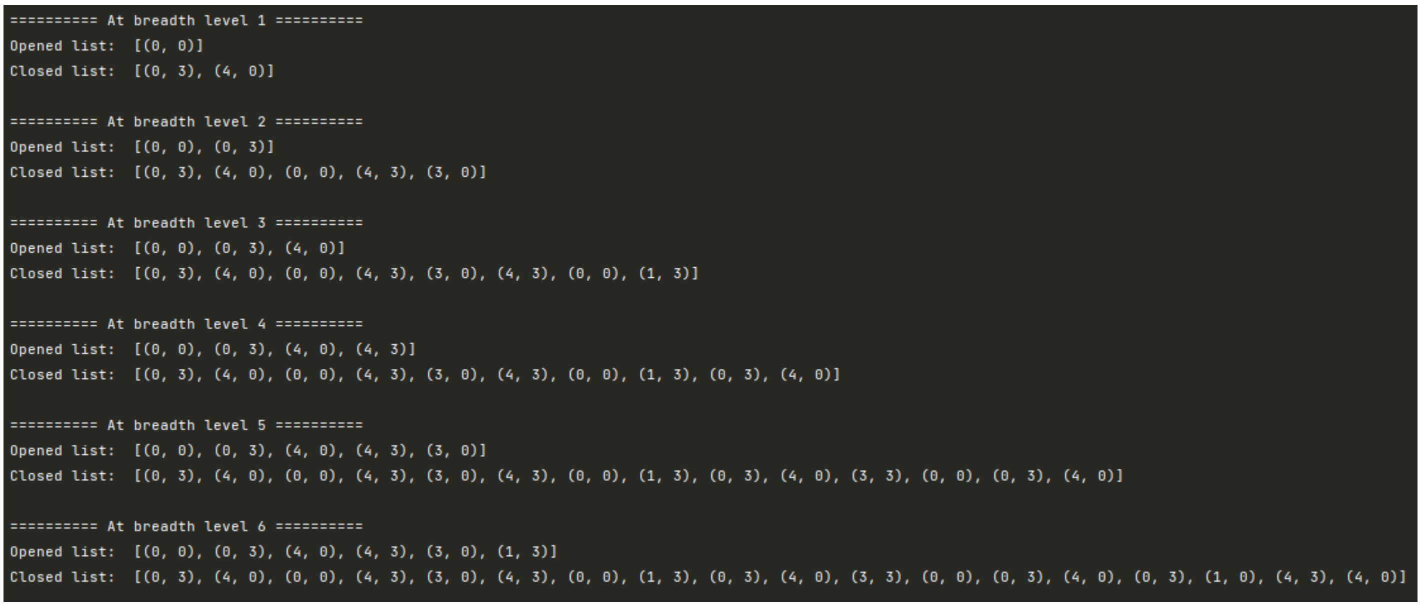
```
LEFT_BUCKET_CAPACITY = 4
RIGHT_BUCKET_CAPACITY =3
GOAL = (0, 2)
RESULT = []

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
```

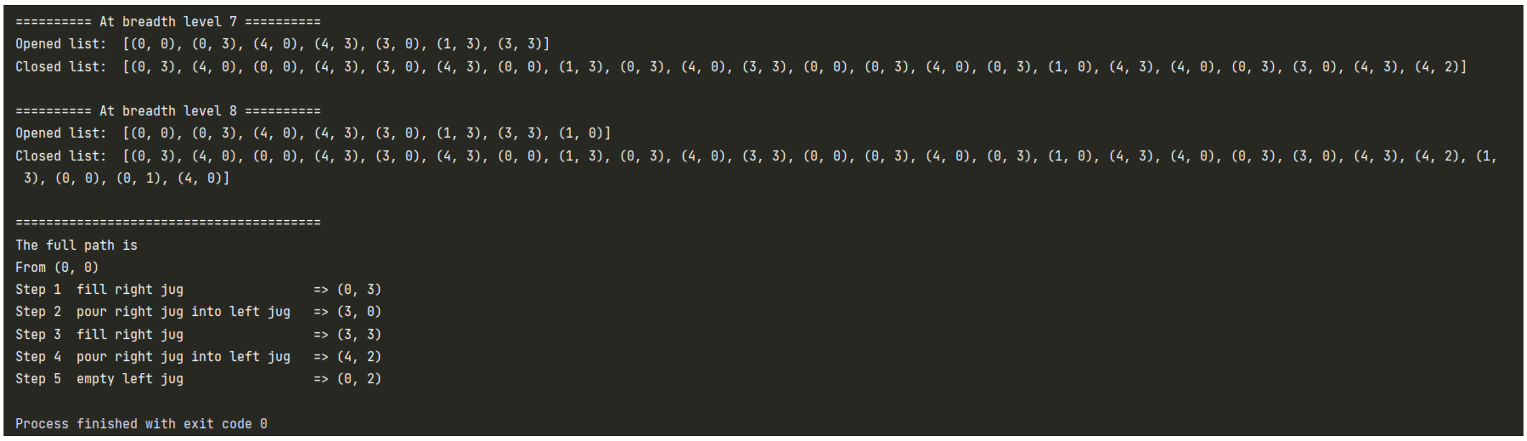
```
 return operations

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:



```
========== At breadth level 1 ==========
Opened list:  [(0, 0)]
Closed list:  [(0, 3), (4, 0)]

========== At breadth level 2 ==========
Opened list:  [(0, 0), (0, 3)]
Closed list:  [(0, 3), (4, 0), (0, 0), (4, 3), (3, 0)]

========== At breadth level 3 ==========
Opened list:  [(0, 0), (0, 3), (4, 0)]
Closed list:  [(0, 3), (4, 0), (0, 0), (4, 3), (3, 0), (4, 3), (0, 0), (1, 3)]

========== At breadth level 4 ==========
Opened list:  [(0, 0), (0, 3), (4, 0), (4, 3)]
Closed list:  [(0, 3), (4, 0), (0, 0), (4, 3), (3, 0), (4, 3), (0, 0), (1, 3), (0, 3), (4, 0)]

========== At breadth level 5 ==========
Opened list:  [(0, 0), (0, 3), (4, 0), (4, 3), (3, 0)]
Closed list:  [(0, 3), (4, 0), (0, 0), (4, 3), (3, 0), (4, 3), (0, 0), (1, 3), (0, 3), (4, 0), (3, 3), (0, 0), (0, 3), (4, 0)]

========== At breadth level 6 ==========
Opened list:  [(0, 0), (0, 3), (4, 0), (4, 3), (3, 0), (1, 3)]
Closed list:  [(0, 3), (4, 0), (0, 0), (4, 3), (3, 0), (4, 3), (0, 0), (1, 3), (0, 3), (4, 0), (3, 3), (0, 0), (0, 3), (4, 0), (0, 3), (1, 0), (4, 3), (4, 0)]
```



```
========== At breadth level 7 ==========
Opened list:  [(0, 0), (0, 3), (4, 0), (4, 3), (3, 0), (1, 3), (3, 3)]
Closed list:  [(0, 3), (4, 0), (0, 0), (4, 3), (3, 0), (4, 3), (0, 0), (1, 3), (0, 3), (4, 0), (3, 3), (0, 0), (0, 3), (4, 0), (0, 3), (1, 0), (4, 3), (4, 0), (0, 3), (3, 0), (4, 3), (4, 2)]

========== At breadth level 8 ==========
Opened list:  [(0, 0), (0, 3), (4, 0), (4, 3), (3, 0), (1, 3), (3, 3), (1, 0)]
Closed list:  [(0, 3), (4, 0), (0, 0), (4, 3), (3, 0), (4, 3), (0, 0), (1, 3), (0, 3), (4, 0), (3, 3), (0, 0), (0, 3), (4, 0), (0, 3), (1, 0), (4, 3), (4, 0), (0, 3), (3, 0), (4, 3), (4, 2), (1,
 3), (0, 0), (0, 1), (4, 0)]

========================================
The full path is
From (0, 0)
Step 1  fill right jug                 => (0, 3)
Step 2  pour right jug into left jug   => (3, 0)
Step 3  fill right jug                 => (3, 3)
Step 4  pour right jug into left jug   => (4, 2)
Step 5  empty left jug                 => (0, 2)

Process finished with exit code 0
```