Lab 4: Water Jug Problem Using BFS & DFS

In [1]:

```
import collections
```

In [2]:

```
#This method return a key value for a given node.
#Node is a list of two integers representing current state of the jugs
def get_index(node):
    return pow(7, node[0]) * pow(5, node[1])
```

In [3]:

```
#This method accepts an input for asking the choice for type of searching required .
#Method return True for BFS, False otherwise
def get_search_type():
    s = input("Enter 'b' for BFS, 'd' for DFS: ")
    #TODO:convert the input into lowercase using lower() method
    s = s[0].lower()

    #TODO: Accept the input again if given input is not as 'b' for BFS, 'd' for DFS
    while s != 'b' and s != 'd':
        s = input("The input is not valid! Enter 'b' for BFS, 'd' for DFS: ")
        s = s[0].lower()
    #TODO: Return True for BFS option selected
    return s == 'b'
```

In [4]:

```
#This method accept volumes of the jugs as an input from the user.
#Returns a list of two integeres representing volumes of the jugs.
def get_jugs():
    print("Receiving the volume of the jugs...")
    #TODO: Create an empty list
    jugs = []

    temp = int(input("Enter first jug volume (>1): "))
    while temp < 1:
        temp = int(input("Enter a valid amount (>1): "))
    #TODO: Append the input quantity of jug into jugs list
    jugs.append(temp)

    temp = int(input("Enter second jug volume (>1): "))
    while temp < 1:
        temp = int(input("Enter a valid amount (>1): "))

    #TODO: Append the input quantity of jug into jugs list
    jugs.append(temp)

    #TODO: Return the list
    return jugs
```

In [5]:

```
#This method accepts the desired amount of water as an input from the user whereas
#the parameter jugs is a list of two integers representing volumes of the jugs
#Returns the desired amount of water as goal
def get_goal(jugs):

    print("Receiving the desired amount of the water...")

    #TODO: Find the maximum capacity of jugs using max()
    max_amount = max(jugs)
    s = "Enter the desired amount of water (1 - {0}): ".format(max_amount)
    goal_amount = int(input(s))
    #TODO: Accept the input again from the user if the bound of goal_amount is outs
    while goal_amount not in range (1, max_amount):
        goal_amount = int(input("Enter a valid amount (1 - {0}): ".format(max_amoun

    #TODO:Return the goal amount of water
    return goal_amount
```

In [6]:

```
#This method checks whether the given path matches the goal node.
#The path parameter is a list of nodes representing the path to be checked
#The goal_amount parameter is an integer representing the desired amount of water
def is_goal(path, goal_amount):

    print("Checking if the goal is achieved...")

    #TODO: Return the status of the latest path matches with the goal_amount of ano
    return path[-1][0] == goal_amount or path[-1][1] == goal_amount
```

In [7]:

```
#This method validates whether the given node is already visited.
#The parameter node is a list of two integers representing current state of the jugs
#The parameter check_dict is   a dictionary storing visited nodes
def been_there(node, check_dict):

    print("Checking if {0} is visited before...".format(node))

    #TODO: Return True whether a given node already exisiting in a dictionary, othe
    return check_dict.get(node[0]) != None
```

In [8]:

```
#This method returns the list of all possible transitions
#The parameter jugs is a list of two integers representing volumes of the jugs
#The parameter path is a list of nodes represeting the current path
#The parameter check_dict is a dictionary storing visited nodes
def next_transitions(jugs, path, check_dict):

    print("Finding next transitions and checking for the loops...")

    #TODO: create an empty list
    result = []
    next_nodes = []
    node = []

    a_max = jugs[0]
    b_max = jugs[1]

    #TODO: initial amount in the first jug using path parameter
    a = path[-1][0]
    #TODO: initial amount in the second jug using path parameter
    b = path[-1][1]

    #Operation Used in Water Jug problem
    # 1. fill in the first jug
    node.append(a_max)
    node.append(b)
    if not been_there(node, check_dict):
        next_nodes.append(node)
    node = []

    # 2. fill in the second jug
    #TODO: Append with the initial amount of water in first jug
    node.append(a)
    #TODO: Append with the max amount of water in second jug
    node.append(b_max)
    #TODO: Check if node is not visited then append the node in next_nodes. Use been
    if not been_there(node, check_dict):
        next_nodes.append(node)
    node = []

    # 3. second jug to first jug
    node.append(min(a_max, a + b))
    node.append(b - (node[0] - a))  # b - ( a' - a)
    if not been_there(node, check_dict):
        next_nodes.append(node)
    node = []

    # 4. first jug to second jug
    #TODO: Append the minimum between the a+b and b_max
    node.append(min(a+b, b_max))
    node.insert(0, a - (node[0] - b))
    if not been_there(node, check_dict):
        next_nodes.append(node)
    node = []

    # 5. empty first jug
    #TODO: Append 0 to empty first jug
    node.append(0)
    #TODO:Append b amount for second jug
    node.append(b)
```

```
    if not been_there(node, check_dict):
        next_nodes.append(node)
    node = []

    # 6. empty second jug
    #TODO:Append a amount for first jug
    node.append(a)
    #TODO: Append 0 to empty second jug
    node.append(0)
    if not been_there(node, check_dict):
        next_nodes.append(node)

    # create a list of next paths
    for i in range(0, len(next_nodes)):
        temp = list(path)
        #TODO: Append the ith index of next_nodes to temp
        temp.append(next_nodes[i])
        result.append(temp)

    if len(next_nodes) == 0:
        print("No more unvisited nodes...\nBacktracking...")
    else:
        print("Possible transitions: ")
        for nnode in next_nodes:
            print(nnode)
    #TODO: return result
    return result
```

In [9]:

```
# This method returns a string explaining the transition from old state/node to new
# The parameter old is a list representing old state/node
# The parameter new is a list representing new state/node
# The parameter jugs is a list of two integers representing volumes of the jugs

def transition(old, new, jugs):

    #TODO: Get the amount of water from old state/node for first Jug
    a = old[0]
    #TODO: Get the amount of water from old state/node for second Jug
    b = old[1]
    #TODO: Get the amount of water from new state/node for first Jug
    a_prime = new[0]
    #TODO: Get the amount of water from new state/node for second Jug
    b_prime = new[1]
    #TODO: Get the amount of water from jugs representing volume for first Jug
    a_max = jugs[0]
    #TODO: Get the amount of water from jugs representing volume for second Jug
    b_max = jugs[1]

    if a > a_prime:
        if b == b_prime:
            return "Clear {0}-liter jug:\t\t\t".format(a_max)
        else:
            return "Pour {0}-liter jug into {1}-liter jug:\t".format(a_max, b_max)
    else:
        if b > b_prime:
            if a == a_prime:
                return "Clear {0}-liter jug:\t\t\t".format(b_max)
            else:
                return "Pour {0}-liter jug into {1}-liter jug:\t".format(b_max, a_ma
        else:
            if a == a_prime:
                return "Fill {0}-liter jug:\t\t\t".format(b_max)
            else:
                return "Fill {0}-liter jug:\t\t\t".format(a_max)
```

In [10]:

```
#This method prints the goal path
#The path is a list of nodes representing the goal path
#The jugs is a list of two integers representing volumes of the jugs

def print_path(path, jugs):

    print("Starting from:\t\t\t\t", path[0])
    for i in  range(0, len(path) - 1):
        print(i+1,":", transition(path[i], path[i+1], jugs), path[i+1])
```

In [11]:

```
#This method searches for a path between starting node and goal node
# The parameter starting_node is a list of list of two integers representing initia
#The parameter jugs a list of two integers representing volumes of the jugs
#The parameter goal_amount is an integer represting the desired amount
#The parameter check_dict is  a dictionary storing visited nodes
#The parameter is_breadth is implements BFS, if True; DFS otherwise
def search(starting_node, jugs, goal_amount, check_dict, is_breadth):

    if is_breadth:
        print("Implementing BFS...")
    else:
        print("Implementing DFS...")

    goal = []
    #TODO: SET accomplished to be False
    accomplished = False

    #TODO: Call a deque() using collections
    q = collections.deque()
    q.appendleft(starting_node)

    while len(q) != 0:
        path = q.popleft()
        check_dict[get_index(path[-1])] = True
        if len(path) >= 2:
            print(transition(path[-2], path[-1], jugs), path[-1])
        if is_goal(path, goal_amount):
            #TODO: Set accomplished to be True
            accomplished = True
            goal = path
            break

        #TODO: Call next_transitions method for generating the further nodes
        next_moves = next_transitions(jugs, path, check_dict)
        #TODO: Iterate over the next_moves list
        for i in next_moves:
            if is_breadth:
                q.append(i)
            else:
                q.appendleft(i)

    if accomplished:
        print("The goal is achieved\nPrinting the sequence of the moves...\n")
        print_path(goal, jugs)
    else:
        print("Problem cannot be solved.")
```

In [12]:

```
if __name__ == '__main__':
    starting_node = [[0, 0]]
    #TODO: Call the get_jugs() method
    jugs = get_jugs()
    #TODO: Call the get_goal() method
    goal_amount = get_goal(jugs)
    #TODO: Create an empty dictionary
    check_dict = {}
    #TODO: call the get_search_type() method
    is_breadth = get_search_type()
    #TODO: Call the search method with the required parameters
    search(starting_node, jugs, goal_amount, check_dict, is_breadth)
```

```
Receiving the volume of the jugs...
Enter first jug volume (>1): 2
Enter second jug volume (>1): 3
Receiving the desired amount of the water...
Enter the desired amount of water (1 - 3): 2
Enter 'b' for BFS, 'd' for DFS: b
Implementing BFS...
Checking if the goal is achieved...
Finding next transitions and checking for the loops...
Checking if [2, 0] is visited before...
Checking if [0, 3] is visited before...
Checking if [0, 0] is visited before...
Checking if [0, 0] is visited before...
Checking if [0, 0] is visited before...
Checking if [0, 0] is visited before...
Possible transitions:
[2, 0]
[0, 3]
[0, 0]
[0, 0]
[0, 0]
[0, 0]
Fill 2-liter jug:                          [2, 0]
Checking if the goal is achieved...
The goal is achieved
Printing the sequence of the moves...

Starting from:                             [0, 0]
1 : Fill 2-liter jug:                      [2, 0]
```