#Lab(1a).Implement BFS using python libraries

```
q=[]

front=-1
rear=-1
def enqueue(k):
    global front,rear
    if front==-1:
        front=rear=0
    else:
        rear+=1
    q.append(k)
def dequeue():
    global front,rear
    k=q[front]
    if front<rear:
        front+=1
    elif front==rear:
        front=-1
        rear=-1
        q.clear()
    return k
s=[]
def BFS(g,n,src):

    vis=[0 for i in range(n)]
    enqueue(src)
    while(front!=-1):
        k=dequeue()
```

```
        if not vis[k]:

            vis[k]=1
            s.append(str(k))
            for i in range(n):
                if g[k][i]==1 and vis[i]==0:
                    enqueue(i)
n=int(input('enter no\' of nodes: '))
print("Enter adjancency matrix: ")
g=eval(input())
#src=int(input('enter source node: '))
for i in range(n):
    BFS(g,n,i)

    print('Sequence: ',' '.join(s))
    s.clear()
```

#Lab(1b).Implement DFS using python libraries

```
s=[]

top=-1

def  push(k):
    global  top
    if top==-1:
        top=0
    else:
        top+=1
    s.insert(top,k)
def pop():
    global top
    k=s[top]
```

```
    top-=1
    return k
t=[]

def DFS(g,n,src):

    vis=[0 for i in range(n)]
    push(src)
    while(top>=0):
        k=pop()

        if vis[k]==0:

            vis[k]=1
            t.append(str(k))
            for i in range(n):
                if g[k][i]==1 and vis[i]==0:
                    push(i)
n=int(input('enter no\' of nodes: '))
print("Enter adjancency matrix: ")
g=eval(input())
#src=int(input('enter source node: '))
for i in range(n):
    DFS(g,n,i)

    print('Sequence: ',' '.join(t))
    t.clear()
```

#2 Finding Maximum depth of the Binary tree using BFS and DFS.

2a) BFS, 2b) BFS

2a)

```
from collections import deque

# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
def maxDepth(root: TreeNode) -> int:
    if not root:
        return 0  # If the tree is empty, depth is 0

    # Use a queue to perform level-order traversal
    queue = deque([root])
    depth = 0
    while queue:
        # Process all nodes at the current level
        level_size = len(queue)
        for _ in range(level_size):

            node = queue.popleft()  # Get the front of the queue
            if node.left:
                queue.append(node.left)  # Add left child to the queue
            if node.right:
                queue.append(node.right)  # Add right child to the queue
        # Increment depth after processing one level
        depth += 1
    return depth
root = TreeNode(1)
root.left = TreeNode(2)
```

```python
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
# Call the function to get the max depth

print("Max Depth of the binary tree:", maxDepth(root))
```

#2B)

```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
def maxDepth(root: TreeNode) -> int:

    # Base case: If the tree is empty, return depth as 0
    if not root:
        return 0

    # Recursively find the depth of left and right subtrees
    left_depth = maxDepth(root.left)
    right_depth = maxDepth(root.right)

    # Return the larger of the two depths, plus 1 for the current node
    return max(left_depth, right_depth) + 1
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)

# Call the function to get the max depth
```

```
print("Max Depth of the binary tree:", maxDepth(root))
```

3. Implement the following uninformed search techniques
    a. Uniform Cost Search
    b. Depth-First Iterative Deepening
    c. Bidirectional

#3A)

```
def UCS(graph, s, goal):
    frontier = {s: 0}
    explored = []
    while frontier:
        print(f"Frontier: {frontier}")
        print(f"Explored: {explored}")
        node = min(frontier, key=frontier.get)
        val = frontier[node]
        print(node, ":", val)
        del frontier[node]
        if goal == node:
            return f"Goal reached with cost: {val}"
        explored.append(node)
        for neighbour, pathCost in graph[node].items():
            if neighbour not in explored and neighbour not in frontier:
                frontier.update({neighbour: val + pathCost})
            elif neighbour in frontier and pathCost > val:
                frontier.update({neighbour: val})
    return "Goal not found"
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'D': 2, 'E': 5},
    'C': {'A': 4, 'F': 3},
```

```
'D': {'B': 2},
'E': {'B': 5, 'F': 2},
```

```
    'F': {'C': 3, 'E': 2}
}
s = input("Enter source node: ")
g = input("Enter goal node: ")
print(UCS(graph, s, g))
```

#3b)

```
def recursiveDLS(graph, v, goal, limit):
    if v == goal:
        return 'GOAL'
    elif limit == 0:
        return 'LIMIT'
    else:
        cutoff = False
        print(v, end=' ')
        for neighbour in graph[v]:
            result = recursiveDLS(graph, neighbour, goal, limit-1)
            if result == 'LIMIT':
                cutoff = True
            elif result != 'FAIL':
                return result
        return 'LIMIT' if cutoff else 'FAIL'
def IDS(graph, v, goal):
    for depth in range(100):
        result = recursiveDLS(graph, v, goal, depth)
        print()
        if result != 'LIMIT':
            return result, depth
```

```python
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': ['G', 'H'],
    'E': [],
    'F': ['I', 'K'],
    'G': [],
    'H': ['L'],
    'I': [],
    'K': ['M'],
    'L': [],
    'M': []
}
s = input("Enter source node: ")
g = input("Enter goal node: ")
res, depth = IDS(graph, s, g)
if res == 'GOAL':
    print("Goal found at depth:", depth)
else:
    print("Goal not found")
```

#3C

```
def BFS(direction, graph, frontier, reached):
    if direction == 'F':
        d = 'c'
    elif direction == 'B':
        d = 'p'
    node = frontier.pop(0)
    for child in graph[node][d]:
        if child not in reached:
            reached.append(child)
            frontier.append(child)
    return frontier, reached
def isIntersecting(reachedF, reachedB):
    intersecting = set(reachedF).intersection(set(reachedB))
    return list(intersecting)[0] if intersecting else -1
def BidirectionalSearch(graph, source, dest):
    frontierF = [source]
    frontierB = [dest]
    reachedF = [source]
    reachedB = [dest]
    while frontierF and frontierB:
        print("From front: ")
        print(f"\tFrontier: {frontierF}")
        print(f"\tReached: {reachedF}")
        print("From back: ")
        print(f"\tFrontier: {frontierB}")
        print(f"\tReached: {reachedB}")

        frontierF, reachedF = BFS('F', graph, frontierF, reachedF)
```

```
        frontierB, reachedB = BFS('B', graph, frontierB, reachedB)
        intersectingNode = isIntersecting(reachedF, reachedB)
        if intersectingNode != -1:
            print("From front: ")
            print(f"\tFrontier: {frontierF}")
            print(f"\tReached: {reachedF}")
            print("From back: ")
            print(f"\tFrontier: {frontierB}")
            print(f"\tReached: {reachedB}")
            print("Path found!")
            path = reachedF[:-1] + reachedB[::-1]
            return path
    print("No path found!")
    return []
def create_graph():
    graph = {}
    n = int(input("Enter number of nodes in graph: "))
    for _ in range(n):
        node = input("Enter node: ")
        children = input(f"Enter children of {node} (comma-separated): ").split(',')
        parents = input(f"Enter parents of {node} (comma-separated): ").split(',')
        graph[node] = {'c': children, 'p': parents}
    return graph
if __name___== "__main__":
    graph = create_graph()
    source = input("Enter source node: ")
    dest = input("Enter destination node: ")
    path = BidirectionalSearch(graph, source, dest)
```

```
if path:
    print(f"Path: {path}")
```

#4a) Water Jug using BFS

```
from collections import deque

def production_jug(action, xCap, yCap, xCur, yCur):
    if action == "fill_x":
        return xCap, yCur
    elif action == "fill_y":
        return xCur, yCap
    elif action == "pour_x":
        return 0, yCur
    elif action == "pour_y":
        return xCur, 0
    elif action == "x_to_y":
        transfer = min(xCur, yCap - yCur)
        return xCur - transfer, yCur + transfer
    elif action == "y_to_x":
        transfer = min(yCur, xCap - xCur)
        return xCur + transfer, yCur - transfer
    return 0, 0


def canMeasureWater(x, y, target):
    # Base cases
    if x + y == target or x == target or y == target:
        return True
    operations = ["fill_x", "fill_y", "pour_x", "pour_y", "y_to_x", "x_to_y"]
    queue = deque([(0, 0)])  # Start with both jugs empty
    visited = set()
    # Perform BFS to explore all possible states
    while queue:
```

```
        x1, y1 = queue.popleft()
        if (x1, y1) in visited:
            continue
        if x1 == target or y1 == target or x1 + y1 == target:
            return True
        visited.add((x1, y1))
        for op in operations:
            next_state = production_jug(op, x, y, x1, y1)
            if next_state not in visited:
                queue.append(next_state)
    return False
xCap = int(input("Enter the capacity of the first jug (x): "))
yCap = int(input("Enter the capacity of the second jug (y): "))
target = int(input("Enter the target amount of water: "))
# Check if it's possible to measure the target water
result = canMeasureWater(xCap, yCap, target)
if result:
    print(f"Yes, it's possible to measure {target} liters of water.")
else:
    print(f"No, it's not possible to measure {target} liters of water.")
```

#4b) Water Jug using DFS

```
def production_jug(action, xCap, yCap, xCur, yCur):
    if action == "fill_x":
        return xCap, yCur
    elif action == "fill_y":
        return xCur, yCap
```

```
    elif action == "pour_x":
        return 0, yCur
    elif action == "pour_y":
        return xCur, 0
    elif action == "x_to_y":
        transfer = min(xCur, yCap - yCur)
        return xCur - transfer, yCur + transfer
    elif action == "y_to_x":
        transfer = min(yCur, xCap - xCur)
        return xCur + transfer, yCur - transfer
    return 0, 0
def canMeasureWater(x, y, target):
    # Base cases
    if x + y == target or x == target or y == target:
        return True
    operations = ["fill_x", "fill_y", "pour_x", "pour_y", "y_to_x", "x_to_y"]
    stack = [(0, 0)]  # Start with both jugs empty
    visited = set()
    # Perform DFS to explore all possible states
    while stack:
        x1, y1 = stack.pop()  # Pop the most recent state (DFS)
        if (x1, y1) in visited:
            continue
        if x1 == target or y1 == target or x1 + y1 == target:
            return True
        visited.add((x1, y1))
        for op in operations:
            next_state = production_jug(op, x, y, x1, y1)
```

```
        if next_state not in visited:
            stack.append(next_state)  # Push the next state onto the stack
    return False

xCap = int(input("Enter the capacity of the first jug (x): "))
yCap = int(input("Enter the capacity of the second jug (y): "))
target = int(input("Enter the target amount of water: "))
# Check if it's possible to measure the target water
result = canMeasureWater(xCap, yCap, target)
if result:
    print(f"Yes, it's possible to measure {target} liters of water.")
else:
    print(f"No, it's not possible to measure {target} liters of water.")
```

5, Missionaries and cannibals

```
from collections import deque

def is_goal(state):
    #left side vi anni right side ki velladam
    return state['left']['missionaries'] == 0 and state['left']['cannibals'] == 0

def is_valid(state):
    for side in ['left', 'right']:
        if state[side]['missionaries'] < 0 or state[side]['cannibals'] < 0:
            return False
        if state[side]['missionaries'] > 0 and state[side]['cannibals'] > state[side]['missionaries']:
            return False
    return True

#fn generating all possible moves
def generate_next_states(state):
    possible_moves = []
    boat_side = state['boat']
    other_side = 'left' if boat_side == 'right' else 'right'
    for missionaries in range(3):
        for cannibals in range(3):
            if 1 <= missionaries + cannibals <= 2:
                move = {'missionaries': missionaries, 'cannibals': cannibals}

                # Check move is valid  or not
                if state[boat_side]['missionaries'] >= move['missionaries'] and
state[boat_side]['cannibals'] >= move['cannibals']:
```

```
                new_state = { 'left': state['left'].copy(), 'right': state['right'].copy(), 'boat': other_side
}

                new_state[boat_side]['missionaries'] -= move['missionaries']
                new_state[boat_side]['cannibals'] -= move['cannibals']
                new_state[other_side]['missionaries'] += move['missionaries']
                new_state[other_side]['cannibals'] += move['cannibals']

                if is_valid(new_state):
                    possible_moves.append(new_state)

        return possible_moves

    # Breadth-First Search (BFS) fn
    def breadth_first_search():
        initial_state = {'left': {'missionaries': 3, 'cannibals': 3}, 'right': {'missionaries': 0, 'cannibals': 0},
'boat': 'left'}
        if is_goal(initial_state):
            return initial_state

        frontier = deque([(initial_state, [])])  # (state, path)
        visited = set()
        visited.add(str(initial_state))

        while frontier:
            current_state, path = frontier.popleft()

            # Check if we've reached the goal
            if is_goal(current_state):
                return path + [current_state]

            next_states = generate_next_states(current_state)
            for next_state in next_states:
                state_str = str(next_state)
                if state_str not in visited:
                    visited.add(state_str)
                    frontier.append((next_state, path + [current_state]))

        return None

    # Depth-First Search (DFS) fn
    def depth_first_search():
        initial_state = {'left': {'missionaries': 3, 'cannibals': 3}, 'right': {'missionaries': 0, 'cannibals': 0},
'boat': 'left'}
        if is_goal(initial_state):
            return initial_state

        frontier = [(initial_state, [])]  # (state, path)
        visited = set()
        visited.add(str(initial_state))

        while frontier:
            current_state, path = frontier.pop()
```

```
        # Check goal reached or not
        if is_goal(current_state):
            return path + [current_state]

        next_states = generate_next_states(current_state)
        for next_state in next_states:
            state_str = str(next_state)
            if state_str not in visited:
                visited.add(state_str)
                frontier.append((next_state, path + [current_state]))

    return None

def print_solution(solution):
    if solution:
        for index, state in enumerate(solution):
            print(f"Step {index + 1}: {state}")
    else:
        print("No solution found.")

print("Missionaries and Cannibals Solution using BFS:")
solution_bfs = breadth_first_search()
print_solution(solution_bfs)
print("\nMissionaries and Cannibals Solution using DFS:")
solution_dfs = depth_first_search()
print_solution(solution_dfs)
```

Outputs:

1A. BFS Implementation Output

Input:

```
Enter no' of nodes: 4
Enter adjacency matrix: [[0,1,1,0],[1,0,0,1],[1,0,0,1],[0,1,1,0]]
```

Output:

```
Sequence: 0 1 2 3
```

1B. DFS Implementation Output

Input:

```
Enter no' of nodes: 4
Enter adjacency matrix: [[0,1,1,0],[1,0,0,1],[1,0,0,1],[0,1,1,0]]
```

Output:

```
Sequence: 0 2 3 1
```

2A. Finding Maximum Depth using BFS

Output:

```
Max Depth of the binary tree: 3
```

2B. Finding Maximum Depth using DFS

Output:

```
Max Depth of the binary tree: 3
```

3A. Uniform Cost Search

Input:

```
Enter source node: A
Enter goal node: F
```

Output:

```
Frontier: {'A': 0}
Explored: []
A : 0
Frontier: {'B': 1, 'C': 4}
Explored: ['A']
B : 1
Frontier: {'C': 4, 'D': 3, 'E': 6}
Explored: ['A', 'B']
D : 3
Frontier: {'C': 4, 'E': 6}
Explored: ['A', 'B', 'D']
C : 4
Frontier: {'E': 6, 'F': 7}
Explored: ['A', 'B', 'D', 'C']
F : 7
Goal reached with cost: 7
```

3B. Depth-First Iterative Deepening

Input:

```
Enter source node: A
Enter goal node: M
```

Output:

```
css
CopyEdit
A
A B
A B D
A B D G
A B D H
A B D H L
A B E
A C
A C F
A C F I
A C F K
A C F K M
Goal found at depth: 4
```

3C. Bidirectional Search

Input:

```
Enter number of nodes in graph: 4
Enter node: A
Enter children of A (comma-separated): B,C
Enter parents of A (comma-separated):
Enter node: B
Enter children of B (comma-separated): D
Enter parents of B (comma-separated): A
Enter node: C
Enter children of C (comma-separated): D
```

```
Enter parents of C (comma-separated): A
Enter node: D
Enter children of D (comma-separated):
Enter parents of D (comma-separated): B,C
Enter source node: A
Enter destination node: D
```

Output:

```
Path found!
Path: ['A', 'B', 'D']
```

4A. Water Jug Problem using BFS

Input:

```
Enter the capacity of the first jug (x): 4
Enter the capacity of the second jug (y): 3
Enter the target amount of water: 2
```

Output:

```
Yes, it's possible to measure 2 liters of water.
```

4B. Water Jug Problem using DFS

Input:

```
Enter the capacity of the first jug (x): 4
Enter the capacity of the second jug (y): 3
Enter the target amount of water: 2
```

Output:

```
Yes, it's possible to measure 2 liters of water.
```

---

5. Missionaries and Cannibals (BFS & DFS)

Output (BFS Solution)

```
Step 1: {'left': {'missionaries': 3, 'cannibals': 3}, 'right':
{'missionaries': 0, 'cannibals': 0}, 'boat': 'left'}
Step 2: {'left': {'missionaries': 2, 'cannibals': 2}, 'right':
{'missionaries': 1, 'cannibals': 1}, 'boat': 'right'}
Step 3: {'left': {'missionaries': 3, 'cannibals': 2}, 'right':
{'missionaries': 0, 'cannibals': 1}, 'boat': 'left'}
Step 4: {'left': {'missionaries': 3, 'cannibals': 0}, 'right':
{'missionaries': 0, 'cannibals': 3}, 'boat': 'right'}
Step 5: {'left': {'missionaries': 3, 'cannibals': 1}, 'right':
{'missionaries': 0, 'cannibals': 2}, 'boat': 'left'}
Step 6: {'left': {'missionaries': 1, 'cannibals': 1}, 'right':
{'missionaries': 2, 'cannibals': 2}, 'boat': 'right'}
Step 7: {'left': {'missionaries': 2, 'cannibals': 1}, 'right':
{'missionaries': 1, 'cannibals': 2}, 'boat': 'left'}
Step 8: {'left': {'missionaries': 0, 'cannibals': 1}, 'right':
{'missionaries': 3, 'cannibals': 2}, 'boat': 'right'}
Step 9: {'left': {'missionaries': 0, 'cannibals': 0}, 'right':
{'missionaries': 3, 'cannibals': 3}, 'boat': 'left'}
```

Output (DFS Solution)

```
csharp
CopyEdit
Same as BFS solution
```