Title: Implement and Demonstrate Depth First Search Algorithm on Water Jug Problem

Introduction to Water jug problem:

Problem Statement:

- Given two jugs with capacities of X liters and Y liters, and an unlimited supply of water, the objective is to measure exactly Z liters using a sequence ofoperations.
- The allowed operations are:
  - Fill either jug completely.
  - Empty either jug.
  - Pour water from one jug into the other until either the first jug is empty or the second jug is full.
- The problem is to determine whether it is possible to obtain exactly Z liters inone of the jugs, and if so, to find the sequence of operations that achieves this.

Constraints:

- All jugs are unmarked (no intermediate measurements).
- The capacities X, Y, and the target Z are positive integers.
- The solution must use only the allowed operations.

Existence of Solution:

- A solution exists if and only if:
- $Z \leq \max(X, Y)$, and
- Z is a multiple of the greatest common divisor (GCD) of X and Y

DFS Pseudocode:

```
function DEPTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a LIFO stack with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the deepest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
```

Python Code:

```python
# Generate all the possible states and transitions for the water jug problem
def generate_graph(jug1, jug2):
    graph = {}
    for x in range(jug1 + 1):
        for y in range(jug2 + 1):
            state = (x, y)
            graph[state] = []
            # Fill jug1
            graph[state].append((jug1, y))
            # Fill jug2
            graph[state].append((x, jug2))
            # Empty jug1
            graph[state].append((0, y))
            # Empty jug2
            graph[state].append((x, 0))
            # Pour jug1 to jug2
            pour_to_jug2 = min(x, jug2 - y)
            graph[state].append((x - pour_to_jug2, y + pour_to_jug2))
            # Pour jug2 to jug1
            pour_to_jug1 = min(y, jug1 - x)
            graph[state].append((x + pour_to_jug1, y - pour_to_jug1))
    return graph

# For the given input generate graph consisting of all the possible states and
transitions
jug1 = 5
jug2 = 3
target = 4
graph = generate_graph(jug1, jug2)

# Implementation of Depth First Search (DFS) to find a path to the target
state
def dfs_graph(graph, start, target):
    stack = [(start, [start])]
    visited = set()
    while stack:
        current, path = stack.pop()
        if current in visited:
            continue
        visited.add(current)
        if target in current:
            return path
        for neighbor in graph[current]:
            if neighbor not in visited:
                stack.append((neighbor, path + [neighbor]))
    return None
```

```python
# Solution path from initial state (0,0) to target state

solution = dfs_graph(graph, (0, 0), target)

print("Solution path:")
for step in solution:
    print(step)
```

Result

```
Solution path:
(0, 0)
(0, 3)
(3, 0)
(3, 3)
(5, 1)
(5, 0)
(2, 3)
(2, 0)
(0, 2)
(5, 2)
(4, 3)
```

Title: Implement and Demonstrate Best First Search Algorithm on Missionaries-Cannibals Problems using Python

Introduction to Missionaries-Cannibals Problem

Problem Statement

- Three missionaries and three cannibals are on one side of a river. They all need to cross to the other side using a boat that can carry at most two people at a time.

Constraints:

- At no point should cannibals outnumber missionaries on either side of the river (or the missionaries will be eaten!).
- The boat cannot cross the river by itself — it needs at least one person to operate it.
- Only missionaries and cannibals are available to row the boat.

BFS Pseudocode:

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTIO N(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        INSERT(child, frontier)
```

Python Code:

```
from collections import deque

# Function to check if a state is valid
# A state is valid if the number of missionaries is not less than the number
of cannibals on both sides of the river.
# m = missionaries, c = cannibals, b = boat position (0 for left bank, 1 for
right bank)

def is_valid(state):
    m, c, b = state
    return (m == 0 or m >= c) and (3 - m == 0 or 3 - m >= 3 - c)

# Generate all possible next states from the current state by moving 1 or 2
people (either missionaries or cannibals) across the river.

def get_next_states(state):
    m, c, b = state
    next_states = []
    if b == 1: # Boat on the starting side
        if m > 0:
            next_states.append((m - 1, c, 0)) # Move 1 missionary
        if m > 1:
            next_states.append((m - 2, c, 0)) # Move 2 missionaries
        if c > 0:
            next_states.append((m, c - 1, 0)) # Move 1 cannibal
        if c > 1:
            next_states.append((m, c - 2, 0)) # Move 2 cannibals
        if m > 0 and c > 0:
            next_states.append((m - 1, c - 1, 0)) # Move 1 missionary and 1
cannibal
    else:  # Boat on the other side
        if m < 3:
            next_states.append((m + 1, c, 1)) # Move 1 missionary
        if m < 2:
            next_states.append((m + 2, c, 1)) # Move 2 missionaries
        if c < 3:
            next_states.append((m, c + 1, 1)) # Move 1 cannibal
        if c < 2:
            next_states.append((m, c + 2, 1)) # Move 2 cannibals
        if m < 3 and c < 3:
            next_states.append((m + 1, c + 1, 1)) # Move 1 missionary and 1
cannibal
    return [state for state in next_states if is_valid(state)]
```

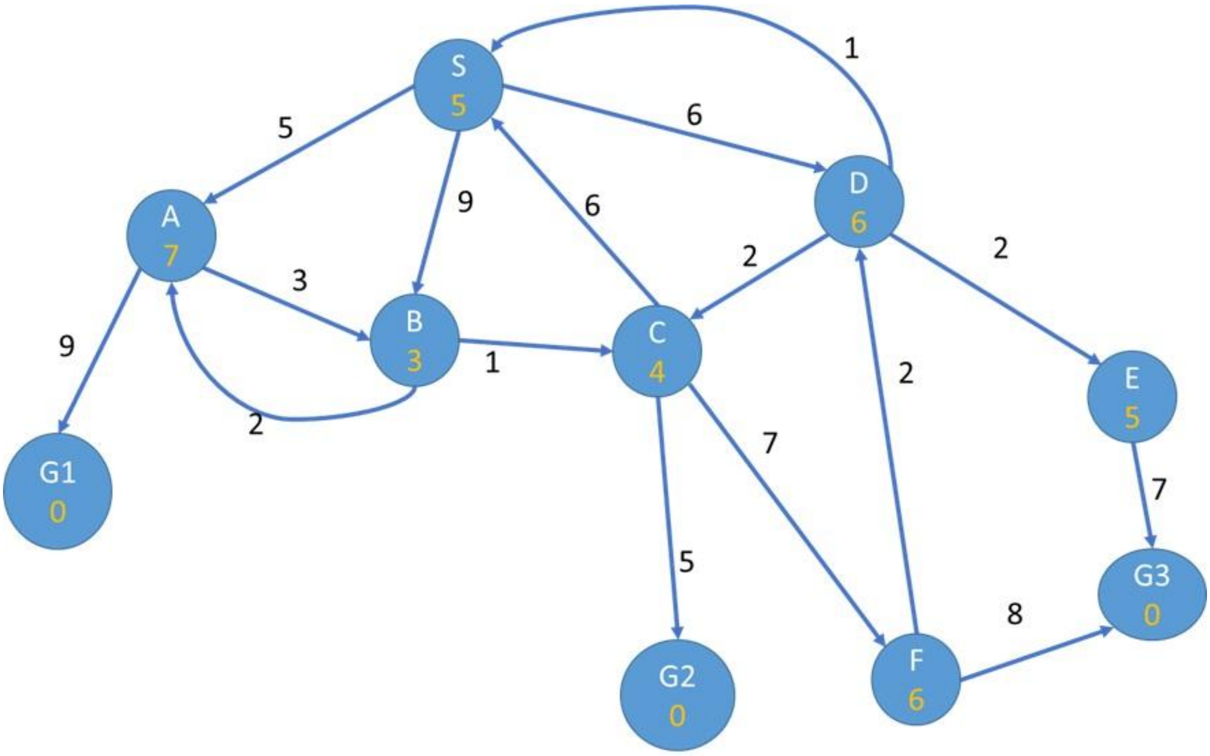
```python
# BFS algorithm to solve the problem
def bfs(initial_state, goal_state):
    queue = deque([(initial_state, [])])
    visited = set()
    while queue:
        state, path = queue.popleft()
        if state == goal_state:
            return path + [state]
        if state not in visited:
            visited.add(state)
            for next_state in get_next_states(state):
                queue.append((next_state, path + [state]))
    return None

# Define the initial state and goal state
# The initial state is (3, 3, 1) representing 3 missionaries, 3 cannibals, and the boat on the starting side.
# The goal state is (0, 0, 0) representing all missionaries and cannibals on the other side.

initial_state = (3, 3, 1)
goal_state = (0, 0, 0)

# Find the solution
solution = bfs(initial_state, goal_state)
if solution:
    for step in solution:
        print(step)
else:
    print("No solution found")
```

Result:

```
(3, 3, 1)
(3, 1, 0)
(3, 2, 1)
(3, 0, 0)
(3, 1, 1)
(1, 1, 0)
(2, 2, 1)
(0, 2, 0)
(0, 3, 1)
(0, 1, 0)
(1, 1, 1)
(0, 0, 0)
```

Title: Implement A* Search algorithm

Example Graph with multiple goals:



A* Pseudocode:

```
function A-STAR-SEARCH(problem) returns a solution, or failure
   node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
   frontier ← a priority queue ordered by f(n) = g(n) + h(n)
   INSERT node into frontier
   explored ← an empty set
   loop do
      if EMPTY?(frontier) then return failure
      node ← POP(frontier)  /* node with lowest f(n) */
      if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
      add node.STATE to explored
      for each action in problem.ACTIONS(node.STATE) do
         child ← CHILD-NODE(problem, node, action)
         if child.STATE is not in explored or frontier then
            INSERT(child, frontier)
         else if child.STATE is in frontier with higher f(child) then
            REPLACE the existing node in frontier with child
```

Python Code:

```
import heapq
import networkx as nx
import matplotlib.pyplot as plt

# A* algorithm for multiple goals
def a_star_multiple_goals(graph, start, goals, heuristic):
    open_list = []
    heapq.heappush(open_list, (heuristic[start], 0, start, [start]))
    closed_set = set()
    best_path = None
    best_cost = float('inf')

    while open_list:
        f, g, current, path = heapq.heappop(open_list)

        if current in goals and g < best_cost:
            best_path = path
            best_cost = g
            continue
        if current in closed_set:
            continue
        closed_set.add(current)

        for neighbor, cost in graph.get(current, []):
            if neighbor not in closed_set:
                new_g = g + cost
                new_f = new_g + heuristic.get(neighbor, float('inf'))
                heapq.heappush(open_list, (new_f, new_g, neighbor, path +
[neighbor]))

    return best_path

# Graph with weights and heuristic values
graph = {
    'S': [('A',5), ('B',9), ('D',6)],
    'A': [('G1',9), ('B',3)],
    'B': [('A',2), ('C',1)],
    'C': [('S',6), ('G2',5), ('F',7)],
    'D': [('S',1), ('E',2), ('C',2)],
    'E': [('G3',7)],
    'F': [('G3',8)],
    'G1': [],
    'G2': [],
    'G3': []
}
```

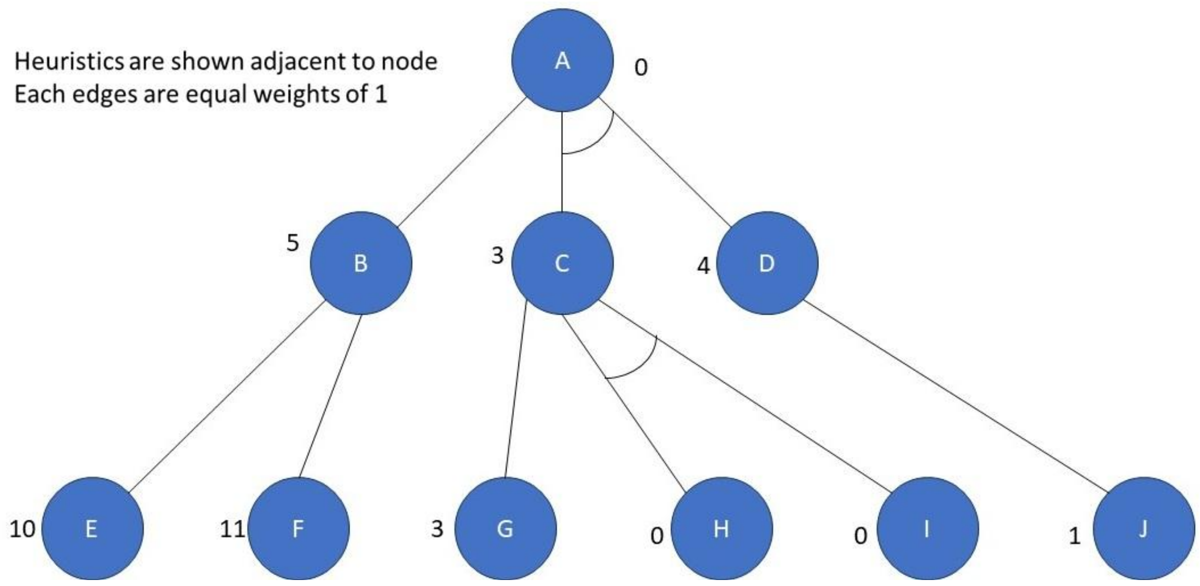
```
heuristic = {
    'A': 7,
    'B': 3,
    'C': 4,
    'D': 6,
    'E': 5,
    'F': 6,
    'S': 5,
    'G1': 0,
    'G2': 0,
    'G3': 0
}

start_node = 'S'
goal_nodes = {'G1', 'G2', 'G3' }

# Find shortest path using A* algorithm
shortest_path = a_star_multiple_goals(graph, start_node, goal_nodes,
heuristic)
print("Shortest path from", start_node, "to one of the goals", goal_nodes,
"is:", shortest_path)
```

Result:

```
Shortest path from S to one of the goals {'G1', 'G3', 'G2'} is: ['S', 'D',
'C', 'G2']
```

Title: Implement AO* Search algorithm

Example Graph



A* Pseudocode:

function AND-OR-SEARCH(problem) returns a conditional plan, or failure
    return OR-SEARCH(problem, problem.INITIAL, [ ])

function OR-SEARCH(problem, state, path) returns a conditional plan, or failure
    if problem.IS-GOAL(state) then return the empty plan
    if IS-CYCLE(path) then return failure
    for each action in problem.ACTIONS(state) do
        plan ←AND-SEARCH(problem, RESULTS(state, action), [state] + path])
        if plan 6= failure then return [action] + plan]
    return failure

function AND-SEARCH(problem, states, path) returns a conditional plan, or failure
    for each s_i in states do
        $plan_i$←OR-SEARCH(problem, s_i, path)
        if $plan_i$ = failure then return failure
    return [if s_1 then $plan_1$ else if s_2 then $plan_2$ else . . . if s_{n-1} then $plan_{n-1}$ else $plan_n$]

Python Code:

```
import networkx as nx
import matplotlib.pyplot as plt

# AO* algorithm function
def ao_star(node, graph, heuristic, solved):
    if node in solved:
        return heuristic[node], [node]

    if not graph[node]:  # Goal node
        solved.add(node)
        return heuristic[node], [node]

    min_cost = float('inf')
    best_path = []

    for child, relation in graph[node]:
        if relation == 'OR':
            cost, path = ao_star(child, graph, heuristic, solved)
            if cost < min_cost:
                min_cost = cost
                best_path = [node] + path
        elif relation == 'AND':
            cost1, path1 = ao_star(child, graph, heuristic, solved)
            siblings = [s for s, r in graph[node] if r == 'AND' and s !=
child]
            total_cost = cost1
            total_path = [node] + path1
            for sib in siblings:
                cost2, path2 = ao_star(sib, graph, heuristic, solved)
                total_cost += cost2
                total_path += path2
            if total_cost < min_cost:
                min_cost = total_cost
                best_path = total_path

    heuristic[node] = min_cost
    solved.add(node)
    return min_cost, best_path

graph = {
    'A' : [('B', 'OR'), ('C', 'AND'), ('D', 'AND')],
    'B' : [('E', 'OR'), ('F', 'OR')],
    'C' : [('G', 'OR'), ('H', 'AND'), ('I', 'AND')],
    'D' : [('J', 'OR')],
    'E' : [],
    'F' : [],
```

```python
    'G' : [],
    'H' : [],
    'I' : [],
    'J' : []
}


heuristic = {
    'A': 0,
    'B': 5,
    'C': 3,
    'D': 4,
    'E': 10,
    'F': 11,
    'G': 3,
    'H': 0,
    'I': 0,
    'J': 1
}

# Run AO* from the Start node
solved_nodes = set()
cost, optimal_path = ao_star('A', graph, heuristic, solved_nodes)

# Output the optimal path and cost
print("Optimal Path:", " -> ".join(optimal_path))
```

Result:

```
Optimal Path: A -> C -> H -> I -> D -> J
```