1. Implement and Demonstrate Depth First Search Algorithm on Water Jug Problem

```
from collections import defaultdict

jug1, jug2, aim = 4, 3, 2

visited = defaultdict(lambda: False)

def waterJugSolverDFS(amt1, amt2):
    if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):
        print(amt1, amt2)
        return True

    visited[(amt1, amt2)] = True
    print(amt1, amt2)

    # Try all possible moves
    if not visited[(0, amt2)] and waterJugSolverDFS(0, amt2):
        return True
    if not visited[(amt1, 0)] and waterJugSolverDFS(amt1, 0):
        return True
    if not visited[(jug1, amt2)] and waterJugSolverDFS(jug1, amt2):
        return True
    if not visited[(amt1, jug2)] and waterJugSolverDFS(amt1, jug2):
        return True
```

```
    # Pour from jug1 to jug2
    pour_amt = min(amt1, jug2 - amt2)
    if not visited[(amt1 - pour_amt, amt2 + pour_amt)] and waterJugSolverDFS(amt1 - pour_amt, amt2 + pour_amt):
        return True

    # Pour from jug2 to jug1
    pour_amt = min(amt2, jug1 - amt1)
    if not visited[(amt1 + pour_amt, amt2 - pour_amt)] and waterJugSolverDFS(amt1 + pour_amt, amt2 - pour_amt):
        return True

    return False

print("Steps:")
waterJugSolverDFS(0, 0)
```

OUTPUT

Steps:

0 0

4 0

4 3

0 3

3 0

3 3

4 2

0 2

2. Implement and Demonstrate Best First Search Algorithm on Missionaries-Cannibals Problems using Python

```
from collections import deque

# Define the initial state
initial_state = {'left': (3, 3), 'right': (0, 0), 'boat': 'left'}

# Define the goal state
goal_state = {'left': (0, 0), 'right': (3, 3), 'boat': 'right'}

# Define a function to check if a state is valid
def is_valid(state):
    left_m, left_c = state['left']
    right_m, right_c = state['right']
    if left_m < 0 or left_c < 0 or right_m < 0 or right_c < 0:
        return False
    if left_m > 3 or left_c > 3 or right_m > 3 or right_c > 3:
        return False
    if left_m < left_c and left_m > 0:
        return False
    if right_m < right_c and right_m > 0:
        return False
    return True
```

```
# Define a function to generate all possible next states from the current state
def generate_next_states(current_state):
    next_states = []
    for i in range(3):
        for j in range(3):
            if i + j > 2 or i + j == 0:
                continue
            if current_state['boat'] == 'left':
                new_state = {
                    'left': (current_state['left'][0] - i, current_state['left'][1] - j),
                    'right': (current_state['right'][0] + i, current_state['right'][1] + j),
                    'boat': 'right'
                }
            else:
                new_state = {
                    'left': (current_state['left'][0] + i, current_state['left'][1] + j),
                    'right': (current_state['right'][0] - i, current_state['right'][1] - j),
                    'boat': 'left'
                }
            if is_valid(new_state):
                next_states.append(new_state)
    return next_states
```

```python
# Define the breadth-first search function
def bfs(initial_state, goal_state):
    visited = set()
    queue = deque([(initial_state, [])])

    while queue:
        current_state, path = queue.popleft()
        if current_state == goal_state:
            return path
        if tuple(current_state['left'] + current_state['right'] + (current_state['boat'],)) in visited:
            continue
        visited.add(tuple(current_state['left'] + current_state['right'] + (current_state['boat'],)))
        for next_state in generate_next_states(current_state):
            queue.append((next_state, path + [next_state]))

    return None
```

```python
# Find the solution using BFS
solution = bfs(initial_state, goal_state)

# Print the solution
if solution:
    print("Solution found with", len(solution), "steps:")
    for i, state in enumerate(solution):
        print("Step", i + 1, ":", state)
else:
    print("No solution found.")
```

Output

Solution found with 11 steps:
Step 1 : {'left': (3, 1), 'right': (0, 2), 'boat': 'right'}
Step 2 : {'left': (3, 2), 'right': (0, 1), 'boat': 'left'}
Step 3 : {'left': (3, 0), 'right': (0, 3), 'boat': 'right'}
Step 4 : {'left': (3, 1), 'right': (0, 2), 'boat': 'left'}
Step 5 : {'left': (1, 1), 'right': (2, 2), 'boat': 'right'}
Step 6 : {'left': (2, 2), 'right': (1, 1), 'boat': 'left'}
Step 7 : {'left': (0, 2), 'right': (3, 1), 'boat': 'right'}
Step 8 : {'left': (0, 3), 'right': (3, 0), 'boat': 'left'}
Step 9 : {'left': (0, 1), 'right': (3, 2), 'boat': 'right'}
Step 10 : {'left': (0, 2), 'right': (3, 1), 'boat': 'left'}
Step 11 : {'left': (0, 0), 'right': (3, 3), 'boat': 'right'}

3. Implement A* Search algorithm

```
def aStarAlgo(start_node, stop_node):
    open_set = set(start_node)
    closed_set = set()
    g = {}               #store distance from starting node
    parents = {}         # parents contains an adjacency map of all nodes
    #distance of starting node from itself is zero
    g[start_node] = 0
    #start_node is root node i.e it has no parent nodes
    #so start_node is set to its own parent node
    parents[start_node] = start_node
    while len(open_set) > 0:
        n = None
        #node with lowest f() is found
        for v in open_set:
            if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):
                n = v
        if n == stop_node or Graph_nodes[n] == None:
            pass
        else:
```

```
            for (m, weight) in get_neighbors(n):
                #nodes 'm' not in first and last set are added to first
                #n is set its parent
                if m not in open_set and m not in closed_set:
                    open_set.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight
                #for each node m,compare its distance from start i.e g(m) to the
                #from start through n node
                else:
                    if g[m] > g[n] + weight:
                        #update g(m)
                        g[m] = g[n] + weight
                        #change parent of m to n
                        parents[m] = n
                        #if m in closed set,remove and add to open
                        if m in closed_set:
                            closed_set.remove(m)
                            open_set.add(m)
        if n == None:
            print('Path does not exist!')
            return None
```

```
        # if the current node is the stop_node
        # then we begin reconstructin the path from it to the start_node
        if n == stop_node:
            path = []
            while parents[n] != n:
                path.append(n)
                n = parents[n]
            path.append(start_node)
            path.reverse()
            print('Path found: {}'.format(path))
            return path
        # remove n from the open_list, and add it to closed_list
        # because all of his neighbors were inspected
        open_set.remove(n)
        closed_set.add(n)
    print('Path does not exist!')
    return None
```

```
#define fuction to return neighbor and its distance
#from the passed node
def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None


        #for simplicity we ll consider heuristic distances given
#and this function returns heuristic distance for all nodes
def heuristic(n):
    H_dist = {
        'A': 11,
        'B': 6,
        'C': 5,
        'D': 7,
        'E': 3,
        'F': 6,
        'G': 5,
        'H': 3,
        'I': 1,
        'J': 0
    }
    return H_dist[n]
```

```
    #Describe your graph here
Graph_nodes = {
    'A': [('B', 6), ('F', 3)],
    'B': [('A', 6), ('C', 3), ('D', 2)],
    'C': [('B', 3), ('D', 1), ('E', 5)],
    'D': [('B', 2), ('C', 1), ('E', 8)],
    'E': [('C', 5), ('D', 8), ('I', 5), ('J', 5)],
    'F': [('A', 3), ('G', 1), ('H', 7)],
    'G': [('F', 1), ('I', 3)],
    'H': [('F', 7), ('I', 2)],
    'I': [('E', 5), ('G', 3), ('H', 2), ('J', 3)],
}

aStarAlgo('A', 'J')
```

Output

```
Path found: ['A', 'F', 'G', 'I', 'J']
['A', 'F', 'G', 'I', 'J']
```

4. Implement AO* Search algorithm

```
# Cost to find the AND and OR path
# Cost to find the AND and OR path
def Cost(H, condition, weight = 1):
    cost = {}
    if 'AND' in condition:
        AND_nodes = condition['AND']
        Path_A = ' AND '.join(AND_nodes)
        PathA = sum(H[node]+weight for node in AND_nodes)
        cost[Path_A] = PathA

    if 'OR' in condition:
        OR_nodes = condition['OR']
        Path_B =' OR '.join(OR_nodes)
        PathB = min(H[node]+weight for node in OR_nodes)
        cost[Path_B] = PathB
    return cost
```

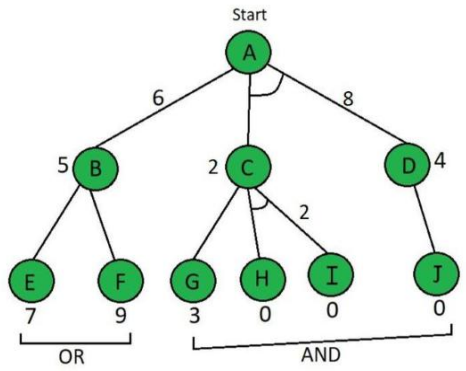
```
# Update the cost
def update_cost(H, Conditions, weight=1):
    Main_nodes = list(Conditions.keys())
    Main_nodes.reverse()
    least_cost= {}
    for key in Main_nodes:
        condition = Conditions[key]
        print(key,':', Conditions[key],'>>>', Cost(H, condition, weight))
        c = Cost(H, condition, weight)
        H[key] = min(c.values())
        least_cost[key] = Cost(H, condition, weight)
    return least_cost
```

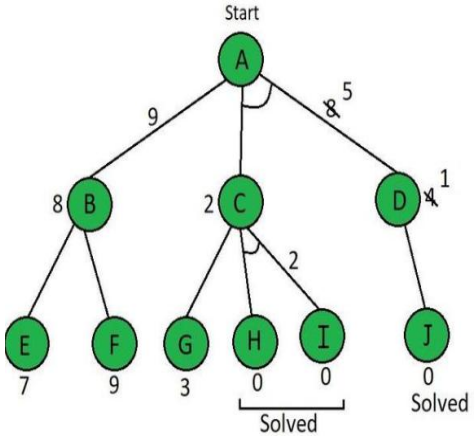
```
# Print the shortest path
def shortest_path(Start,Updated_cost, H):
    Path = Start
    if Start in Updated_cost.keys():
        Min_cost = min(Updated_cost[Start].values())
        key = list(Updated_cost[Start].keys())
        values = list(Updated_cost[Start].values())
        Index = values.index(Min_cost)

        # FIND MINIMIMUM PATH KEY
        Next = key[Index].split()
        # ADD TO PATH FOR OR PATH
        if len(Next) == 1:

            Start =Next[0]
            Path += '<--' +shortest_path(Start, Updated_cost, H)
        # ADD TO PATH FOR AND PATH
        else:
            Path +='<--('+key[Index]+') '

            Start = Next[0]
            Path += '[' +shortest_path(Start, Updated_cost, H) + ' + '

            Start = Next[-1]
            Path += shortest_path(Start, Updated_cost, H) + ']'

    return Path
```

```
H = {'A': -1, 'B': 5, 'C': 2, 'D': 4, 'E': 7, 'F': 9, 'G': 3, 'H': 0, 'I':0, 'J':0}

Conditions = {
'A': {'OR': ['B'], 'AND': ['C', 'D']},
'B': {'OR': ['E', 'F']},
'C': {'OR': ['G'], 'AND': ['H', 'I']},
'D': {'OR': ['J']}
}
# weight
weight = 1
# Updated cost
print('Updated Cost :')
Updated_cost = update_cost(H, Conditions, weight=1)
print('*'*75)
print('Shortest Path :\n',shortest_path('A', Updated_cost,H))
```

Output

```
Updated Cost :
D : {'OR': ['J']} >>> {'J': 1}
C : {'OR': ['G'], 'AND': ['H', 'I']} >>> {'H AND I': 2, 'G': 4}
B : {'OR': ['E', 'F']} >>> {'E OR F': 8}
A : {'OR': ['B'], 'AND': ['C', 'D']} >>> {'C AND D': 5, 'B': 9}
***************************************************************************
Shortest Path :
 A<--(C AND D) [C<--(H AND I) [H + I] + D<--J]
```





5. Solve 8-Queens Problem with suitable assumptions

```
# Taking number of queens as input from user
print ("Enter the number of queens")
N = int(input())

# here we create a chessboard
# NxN matrix with all elements set to 0
board = [[0]*N for _ in range(N)]

def attack(i, j):
    #checking vertically and horizontally
    for k in range(0,N):
        if board[i][k] ==1 or board[k][j] ==1:
            return True
    #checking diagonally
    for k in range(0,N):
        for l in range(0,N):
            if (k+l==i+j) or (k-l==i-j):
                if board[k][l]==1:
                    return True
    return False
```

```
def N_queens(n):
    if n==0:
        return True
    for i in range(0,N):
        for j in range(0,N):
            if (not(attack(i,j))) and (board[i][j]!=1):
                board[i][j] = 1
                if N_queens(n-1)==True:
                    return True
                board[i][j] = 0

    return False

N_queens(N)
for i in board:
    print (i)
```

Output

```
Enter the number of queens
 8
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
```