Experiment No. 2

Name : Aniket Narbariya

UID : 2019130043

Class : TE COMPS Batch C

Subject : AIML

Aim : Implement intelligent agent using bfs and dfs techniques.

Theory :

SEARCH STRATEGIES :

In order to perform a systematic search, the two fundamental strategies are breadth-first and depth-first. In addition, the agent needs to select among nodes at the same level in a systematic manner. The most frequently used selections here are left-to-right or right-to-left with respect to the way the nodes are listed in a graphical view. Depending on the type of the nodes, there is also often an obvious ordering, such as alphabetical or numerical. In this task, use a clockwise ordering starting from the "North" position (i.e. in the order North, East, South, West).

BREADTH-FIRST SEARCH (BFS) ALGORITHM:

In this strategy, all direct successor nodes of a given level are explored first, and then the successor nodes of the first node at this level are examined. In the tree, this means that all nodes at a given level are explored before proceeding to the next level.

DEPTH-FIRST SEARCH (DFS) ALGORITHM :

This search strategy picks the first successor node of the current node, then the first successor node of that node, and so on. In the tree, this means that all nodes of a branch are explored until a leaf node is reached. Once a leaf node is reached, the algorithm "backtracks" (goes up one level), and selects the next node at that level as the new starting point of a sub-branch.

Program :

BFS:

```
from collections import deque
def BFS(a, b, target):

    # Map is used to store the states
    m = {}
    isSolvable = False
    path = []

    # Queue to maintain states
    q = deque()

    # Initialing with initial state
    q.append((0, 0))

    # Iterate till our queue is not empty
    while (len(q) > 0):

        # Take the current state
        u = q.popleft()
        # Continue if this state is already visited
        if ((u[0], u[1]) in m):
            continue

        if ((u[0] > a or u[1] > b or u[0] < 0 or u[1] < 0)):
            continue

        path.append([u[0], u[1]])

        # Marking current state as visited
        m[(u[0], u[1])] = 1

        # If we reach solution state, put ans=1
        if (u[0] == target or u[1] == target):
            isSolvable = True

            if (u[0] == target):
                if (u[1] != 0):
                    path.append([u[0], 0])
            else:
                if (u[0] != 0):
                    path.append([0, u[1]])

            # Print the solution path
            sz = len(path)
            for i in range(sz):
                print("(", path[i][0], ",",path[i][1], ")")
            break

        q.append([u[0], b])
        q.append([a, u[1]])

        for ap in range(max(a, b) + 1):

            # Pour current amount of water from Jug2 to Jug1
            c = u[0] + ap
            d = u[1] - ap

            if (c == a or (d == 0 and d >= 0)):
                q.append([c, d])

            # Pour current amount of water from Jug 1 to Jug2
            c = u[0] - ap
            d = u[1] + ap

            # Here,we check if this state is possible or not
            if ((c == 0 and c >= 0) or d == b):
                q.append([c, d])

        # Empty Jug2
        q.append([a, 0])
        # Empty Jug1
        q.append([0, b])

    # No, solution exists if ans=0
    if (not isSolvable):
        print ("No solution")

l = list(map(int, input("Enter jug capacities ").split()))
jug1 = l[0]
jug2 = l[1]
target_cap = 2
print("Path for solution state is:")
BFS(jug1,jug2,target_cap)
```

DFS:

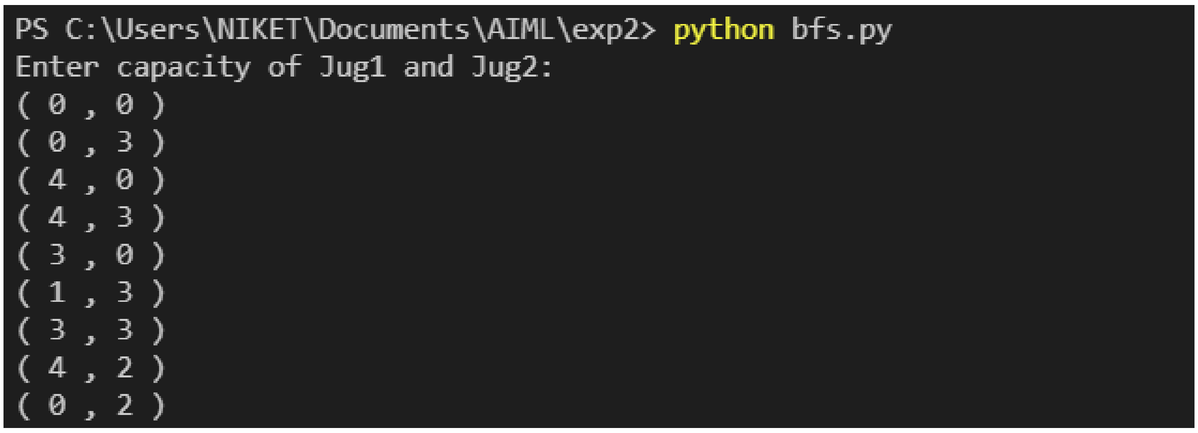
```
def match(p,x,y):
    global c1
    global c2
    if(p[0]==0 and p[1]==0):
        if(x==c1 and y==0):
            return True
        else:
            return False
    elif(p[0]!=0 and p[1]==c2):
        if(x==p[0] and y==0):
            return True
        else:
            return False
    elif(p[0]!=0 and p[1]>=0):
        temp=min(c2-p[1],p[0])
        right=p[1]+temp
        left=p[0]-temp
        # print(x,y)
        if(right==y and left==x):
            return True
        return False
    elif(p[0]==0 and p[1]>=0):
        if(x==c1 and y==p[1]):
            return True
        return False
    return False

def dfs(p):
    if(p[0]==2):
        return
    for i in range(0,5):
        for j in range(0,4):
            if(match(p,i,j)):
                p[0],p[1]=i,j
                print(i,j)
                dfs(p)
            else:
                continue
    return 1

a,b=0,0
c1=int(input('Enter the capacity of 1st jug :'))
c2=int(input('Enter the capacity of 2nd jug :'))
initial_point=[0,0]
print(*initial_point)
dfs(initial_point)
```

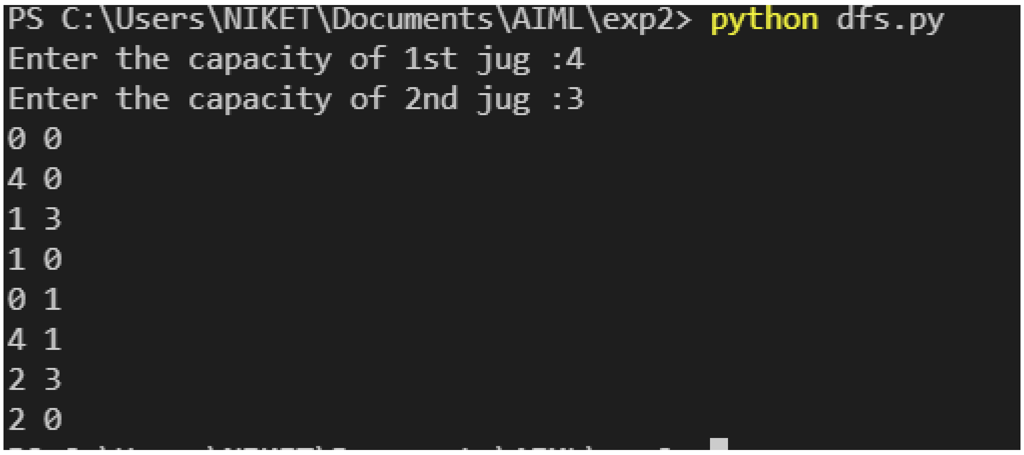
Output:

BFS:



```
PS C:\Users\NIKET\Documents\AIML\exp2> python bfs.py
Enter capacity of Jug1 and Jug2:
( 0 , 0 )
( 0 , 3 )
( 4 , 0 )
( 4 , 3 )
( 3 , 0 )
( 1 , 3 )
( 3 , 3 )
( 4 , 2 )
( 0 , 2 )
```

DFS:



```
PS C:\Users\NIKET\Documents\AIML\exp2> python dfs.py
Enter the capacity of 1st jug :4
Enter the capacity of 2nd jug :3
0 0
4 0
1 3
1 0
0 1
4 1
2 3
2 0
```

Conclusion :

We were able to implement intelligent agent through BFS and DFS technqiues.

In BFS it uses queue data structure and search breadth -wise , while, in DFS we use stack data structure and search using depth-wise(also it is recursive in nature).

Also,here in the water jug problem, as the state space tree is infinite because of loops (which is due to filling and emptying of jugs) thus it useful to do a graph search and maintain the visited nodes to avoid repetition.

Also, in general BFS gives shallowest (most efficient solution) while DFS does not necessarily give the shallowest solution (or any solution at all).