Experiment 2

Objective

To be able to model a given problem in terms of state space search problem and solve the same using BFS/ DFS.

Problem Statement: A

In the rabbit leap problem (see the figure given below), three east-bound rabbits stand in a line blocked by three west-bound rabbits. They are crossing a stream with stones placed in the east-west direction in a line. There is one empty stone between them. The rabbits can only move forward one or two steps. They can jump over one rabbit if the need arises, but not more than that. Are they smart enough to cross each other without having to step into the water?

Problem Statement: B

The missionaries and cannibals' problem is usually stated as follows (see the figure given below). Three missionaries and three cannibals are on one side of a river, along with a boat that can hold one or two people. Find a way to get everyone to the other side without ever leaving a group of missionaries in one place outnumbered by the cannibals in that place. This problem is famous in AI because it was the subject of the first paper that approached problem formulation from an analytical viewpoint.

Lab Exercise

Program Code

1.First write the program for Depth-FirstSearch (DFS) and Breadth-First Search (BFS).

```
Input: s as the source node
BFS (G, s)
let Q be queue.
Q.enqueue( s )
mark s as visited
while ( Q is not empty)
v = Q.dequeue( )

for all neighbors w of v in Graph G
if w is not visited
Q.enqueue( w )
mark w as visited

DFS(graph, start_vertex):
    stack = [start_vertex]
    visited = []

    while stack:
        vertex = stack.pop()

        if vertex not in visited:
            visited.append(vertex)
            for neighbor in graph[vertex]:
                stack.append(neighbor)
    return visited
```

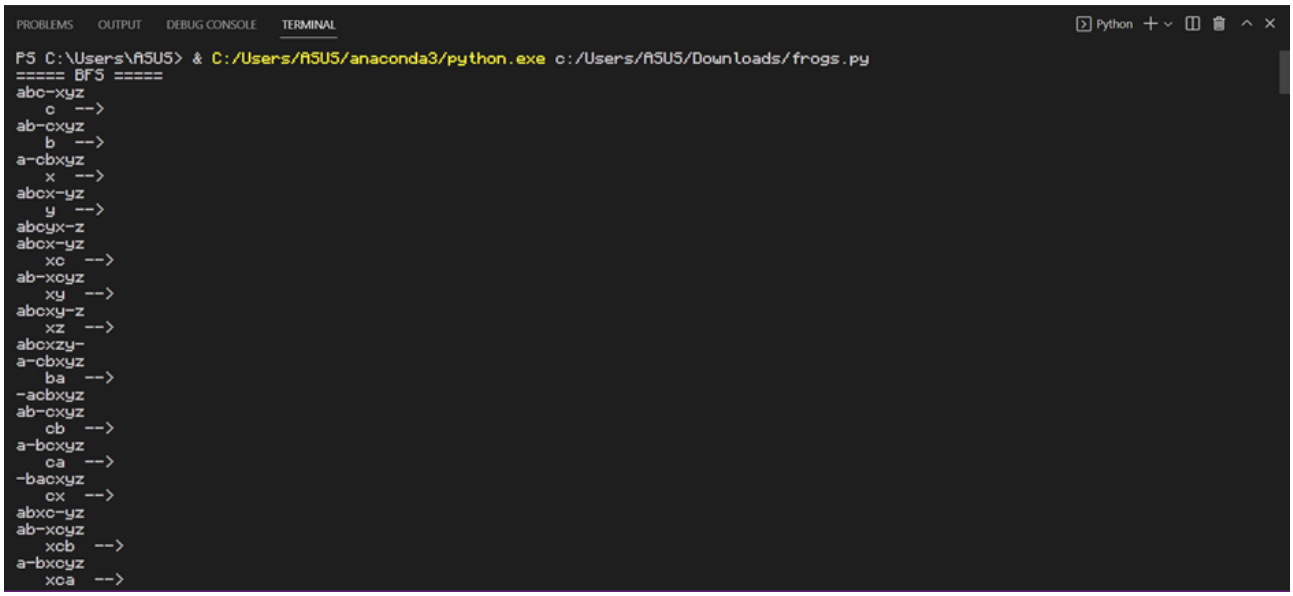
2.Write the programs to solve the given problems using DFS and BFS. Problem A

```
START = 'abc-xyz'
GOAL  = 'xyz-abc'
RR = 'abc'
LR = 'xyz'

def swap_positions(s,i,j):
    assert i < j  #if i < j is TRUE then run the program otherwise stop execution
    return s[:i] + s[j] + s[i+1:j] + s[i] + s[j+1:]

def apply_move(s,f):
    i = s.index(f)
    if f in RR: # go right
```

```
        if i+1 < len(s) and s[i+1] == '-':
            return swap_positions(s,i,i+1)
        if i+2 < len(s) and s[i+1] != '-' and s[i+2] == '-':
            return swap_positions(s,i,i+2)
    if f in LR: # go left
        if i-1 >= 0 and s[i-1] == '-':
            return swap_positions(s,i-1,i)
        if i-2 >= 0 and s[i-1] != '-' and s[i-2] == '-':
            return swap_positions(s,i-2,i)

def show_state(s):
    print (s)

def next_moves(s):
    i = s.index('-')
    ms = []
    # in from the left
    if i-1 >= 0 and s[i-1] in RR:
        ms.append(s[i-1])
    if i-2 >= 0 and s[i-2] in RR:
        ms.append(s[i-2])
    # in from the right
    if i+1 < len(s) and s[i+1] in LR:
        ms.append(s[i+1])
    if i+2 < len(s) and s[i+2] in LR:
        ms.append(s[i+2])
    return ms

def possible(s):
    i = s.index('-')
    if i==0:
        if s[i+2] in RR:
            return False
        else:
            return True
    elif i==len(s):
        if s[i-2] in LR:
            return False
        else:
            return True
    else:
        return True

def search(s):
    work = set([("",s)])
    states = 0
    while work:
        path, cur = work.pop()
        states += 1
        show_state(cur)
        for k in next_moves(cur):
            succ = apply_move(cur, k)
            work.add((path+k, succ))
            print ("  ", path+k, " --> ",)
            show_state(succ)
            if succ == GOAL:
                print ("           !!! GOAL in", len(path)+1, " moves.")
    print ("Visited", states, "states.")

print("===== BFS =====")
search(START)

def dfs2(s,visited=[],path=[]):

    if s==GOAL:
        #print(visited,count,len(visited))
        print ("           !!! GOAL in", len(path), " moves.")
        # [print(item) for item in visited]
        result = ''.join(str(item) for item in path)
        print(result)
        print ("\nVisited", len(visited), "states.")
        quit()
    else:
        for x in next_moves(s):
            po = possible(apply_move(s,x))
            if po==False:
```

```
                continue

            if apply_move(s,x) not in visited:
                print ("  ", path, " --> ",)
                print(apply_move(s,x))
                #print(visited)
                path.append(x)
                visited.append(apply_move(s,x))
                dfs2(apply_move(s,x),visited,path)
                path.pop()

print("====== DFS ======")
dfs2(START,visited=[],path=[])
```

Output



```
Visited 109 states.
====== DFS ======
   []  -->
ab-cxyz
   ['c']  -->
a-bcxyz
   ['c']  -->
abxc-yz
   ['c', 'x']  -->
abx-cyz
   ['c', 'x', 'c']  -->
a-xbcyz
   ['c', 'x', 'c', 'b']  -->
-axbcyz
   ['c', 'x', 'c', 'b', 'a']  -->
xa-bcyz
   ['c', 'x', 'c', 'b', 'a', 'x']  -->
x-abcyz
   ['c', 'x', 'c', 'b']  -->
ax-bcyz
   ['c', 'x', 'c']  -->
abxyc-z
   ['c', 'x', 'c', 'y']  -->
abxy-cz
   ['c', 'x', 'c', 'y', 'c']  -->
abxyzc-
   ['c', 'x', 'c', 'y', 'c', 'z']  -->
abxyz-c
   ['c', 'x', 'c', 'y']  -->
abxycz-
```

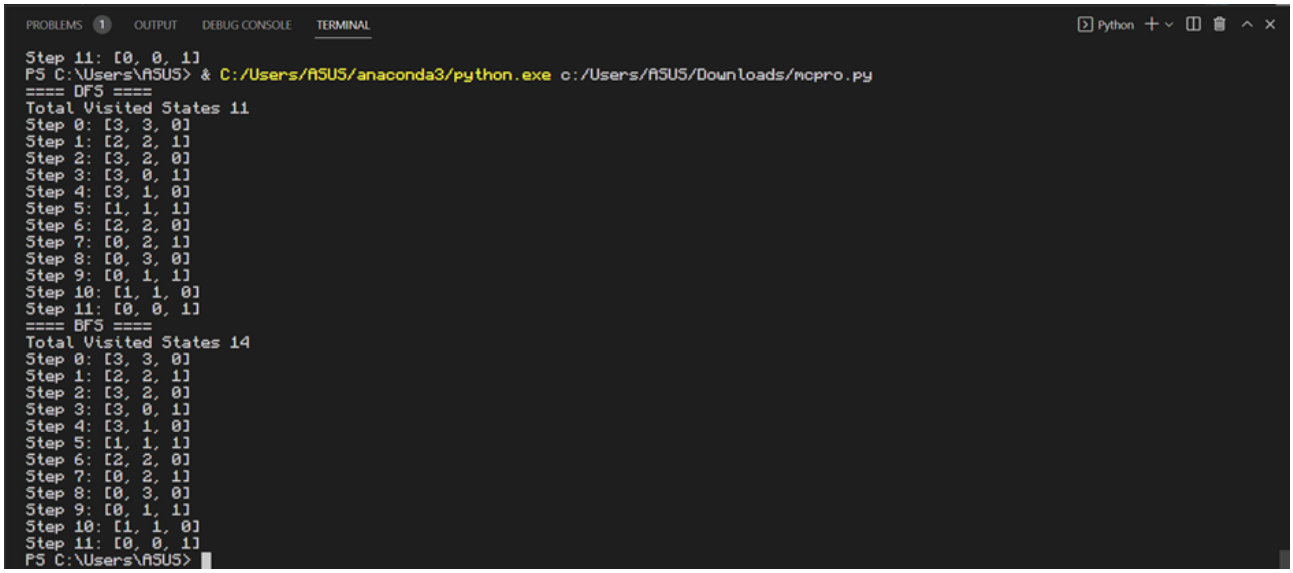
```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL                    Python

abxyc-z
   ['c', 'x', 'c', 'y']  -->
abxy-cz
   ['c', 'x', 'c', 'y', 'c']  -->
abxyzc-
   ['c', 'x', 'c', 'y', 'c', 'z']  -->
abxyz-c
   ['c', 'x', 'c', 'y']  -->
abxycz-
   ['c', 'x', 'c', 'y', 'z']  -->
abxy-zc
   ['c', 'x']  -->
abxcy-z
   ['c', 'x', 'y']  -->
abx-ycz
   ['c', 'x', 'y', 'c']  -->
a-xbycz
   ['c', 'x', 'y', 'c', 'b']  -->
xy-azbc
   ['c', 'x', 'y', 'c', 'b', 'a', 'x', 'y', 'z', 'c', 'b', 'a', 'y']  -->
xyza-bc
   ['c', 'x', 'y', 'c', 'b', 'a', 'x', 'y', 'z', 'c', 'b', 'a', 'y', 'z']  -->
xyz-abc
          !!! GOAL in 15  moves.
cxycbaxyzcbayza

Visited 32 states.
PS C:\Users\ASUS>
```

Problem B

```python
#s[0,1,2] =  m ,c, b

def is_legal(s):

    if s[0]<0 or s[0]>3 or s[1]<0 or s[1]>3:
        return False
    if s[0]>0 and s[0]<s[1]:
        return False
    if (3-s[0])<0 or (3-s[0])>3 or (3-s[1])<0 or (3-s[1])>3:
        return False
    if (3-s[0])>0 and (3-s[0])<(3-s[1]):
        return False

    return True

def is_goal(s):
    if s == [0,0,1]:
        return True
    return False

def is_possible(s):
    q=[]
    state = [[1,0],[0,1],[1,1],[2,0],[0,2]]
    for item in state:
        if s[2]==0:
            curr=[s[0]-item[0],s[1]-item[1],1]
            if is_legal(curr):
                q.append(curr)
        else:
            curr = [s[0]+item[0],s[1]+item[1],0]
```

```
            if is_legal(curr):
                q.append(curr)

    return q

def bfs(s,states,q=[[3,3,0]],visited=set(),parent={}):
    for item in q:
        source = str(item)
        if is_goal(item):
            ans = []
            temp = source
            while temp!= str(s):
                ans.append(temp)
                temp = parent.get(temp)
            return ans,visited

        for state in states:
            if item[2]==0:
                next = [item[0]-state[0],item[1]-state[1],1]
                if is_legal(next):
                    if str(next) not in visited:
                        visited.add(str(next))
                        q.append(next)
                        parent.update({str(next):source})
            else:
                next = [item[0]+state[0],item[1]+state[1],0]
                if is_legal(next):
                    if str(next) not in visited:
                        visited.add(str(next))
                        q.append(next)
                        parent.update({str(next):source})

def dfs(s,start,visited=[],parent={}):
    visited.append(str(s))

    if is_goal(s):
        return [visited,True]

    child = is_possible(s)
    for c in child:
        if str(c) not in visited:
            x, y = dfs(c,start,visited,parent)
            if y == True:
                return [x,True]
            visited.remove(str(c))
    return [[], False]


print("==== DFS ====")
possible_states = [[1,0],[0,1],[1,1],[2,0],[0,2]]
s = [3,3,0]
path1,v = dfs(s,s,visited=[],parent={})
print("Total Visited States",len(path1)-1)
for x,y in enumerate(path1):
     print(f"Step {x}: {y}")

print("==== BFS ====")
path,visit=bfs(s,possible_states,q=[[3,3,0]],visited=set(),parent={})
print("Total Visited States",len(visit)-1)
path.append(s)
path.reverse()
for x,y in enumerate(path):
    print(f"Step {x}: {y}")
```

Output



3.Compare the complexities of both searching techniques.
BFS and DFS are two algorithms that we use for graph traversal. In both the algorithms we start from a node and iterate over the whole graph with V nodes and E edges. Also, the time complexity of iterating over the graph with these traversal techniques is O(V + E).

Conclusion

From LAB 1 and LAB 2 experiment, we learn about BFS and DFS state space searching algorithm. We solved rabbit leap problem, three east-bound rabbits stand in a line blocked by three west-bound rabbits. In BFS, before reaching to the goal state 109 states are visited, while in DFS only 32 states are visited. DFS is more optimal than BFS. We solved Three missionaries and three cannibals' problem. In BFS, before reaching to the goal state 14 states are visited, while in DFS only 11 states are visited. Which proves that DFS is more optimal than BFS. In these two problems is the goal state is listed in depth, So DFS requires less time to solve these problems but it required more Space complexity, In BFS it take more time to solve this type problem rather than DFS because goal state listed at depth. But BFS take less Space complexity than DFS.