Farmer, Goat, Wolf and Cabbage puzzle

using state space method

---

1. **State representation: [Farmer , Goat , Wolf , Cabbage]**

- **value:**
  - **1 -> Left bank**
  - **0 -> Right bank**

1. **Initial state : "1111"**
2. **Final state : "0000"**

Representing states

In [1]:

```
#indices in array repsentation
FARMER = 0
GOAT = 1
WOLF = 2
CABBAGE = 3

START_STATE = "1111"
FINAL_STATE = "0000"
```

To Check valid state

In [2]:

```
def validState(state):
    #if (Goat and Wolf) or (Goat and Cabbage) are not on the farmers side
    if( (state[FARMER]!=state[GOAT] and state[GOAT]==state[WOLF]) or (state[FARMER]!=sta
te[GOAT] and state[GOAT]==state[CABBAGE]) ):
        return False
    else:
        return True
```

To move from one bank to another

In [3]:

```
def move(obj , state):
    if state[obj]=='0':
        s='1'
    else:
        s='0'
    return state[:obj]+s+state[obj+1:]
```

To get possible valid states from current state

In [4]:

```
def nextStates(state):
    possibleValidStates = []
```

```
    #if farmer can move alone
    nextState = move(FARMER , state)
    if(validState(nextState)):
        possibleValidStates.append(nextState)

    #find possible things that a farmer can carry
    for i in range(1,4):
        if(state[FARMER] == state[i]):
            nextState = move(FARMER , state)
            nextState = move(i , nextState)
            if(validState(nextState)):
                possibleValidStates.append(nextState)
    return possibleValidStates
```

To build state space graph and searching algorithm

In [5]:

```
class stateSpaceGraph:
    def __init__(self):
        self.nodeDict = dict() #adj list
        self.treeNodes = set() #visited node list to build tree

    #recc function to build a tree
    def buildTree(self,state):
        if state not in self.treeNodes:
            self.treeNodes.add(state)
            adjStates=nextStates(state)
            for i in adjStates:
                s.addEdge(state,i)
            for i in adjStates:
                self.buildTree(i)

    def addEdge(self,fromNode,toNode):
        #if visiting for the first time
        if fromNode not in self.nodeDict:
            self.nodeDict[fromNode]=[]
        #add to adj list
        self.nodeDict[fromNode].append(toNode)

    #to print state space graph
    def __str__(self):
        return (str(self.nodeDict))

    #BFS searching
    def BFS(self):
        q=[(START_STATE,[START_STATE])]
        visited = set()
        while(s):
            (vertex,path) = q.pop(0)
            if vertex not in visited:
                if vertex == FINAL_STATE:
                    return path
                visited.add(vertex)
                for i in self.nodeDict[vertex]:
                    q.append((i,path+[i]))

    #DFS searching
    def DFS(self):
        s=[(START_STATE,[START_STATE])]
        visited = set()
        while(s):
            (vertex,path) = s.pop()
            if vertex not in visited:
                if vertex == FINAL_STATE:
                    return path
                visited.add(vertex)
                for i in self.nodeDict[vertex]:
                    s.append((i,path+[i]))
```

```
s=stateSpaceGraph()
s.buildTree(START_STATE)
```

State space garph

In [6]:

```
print(s)
```

```
{'1111': ['0011'], '0011': ['1011', '1111'], '1011': ['0011', '0001', '0010'], '0001': ['
1101', '1011'], '1101': ['0001', '0100'], '0100': ['1100', '1110', '1101'], '1100': ['010
0', '0000'], '0000': ['1100'], '1110': ['0010', '0100'], '0010': ['1110', '1011']}
```

Solution

DFS

In [7]:

```
print(s.DFS())
```

```
['1111', '0011', '1011', '0010', '1110', '0100', '1100', '0000']
```

BFS

In [8]:

```
print(s.BFS())
```

```
['1111', '0011', '1011', '0001', '1101', '0100', '1100', '0000']
```