```python
# Coded By Parsa Yousefi Nejad
# Second Project: River Crossing Problem
# Version 4: Iterative_DLS() replaced Recursive DLS_Search() and some minor changes were made
# Add more Comments

# Importing Necessary libraries
from os import system, name  # For Clear() method
from copy import copy        # To shallow copy of an object
from time import sleep       # For implementing pause mechanism in ShowPath()

# Show: Shows one state Graphically
def Show(state):
    # List of Character names to display
    listOfChars = ["POLICE", "THIEF", "FATHER", "MOTHER",
                   "DAUGHTER_1", "DAUGHTER_2", "SON_1", "SON_2",]
    # Shore graphic representation
    shore = ('\x1b[0;32;42m'+"⥮"+'\x1b[0m') * 10

    # Plain text with Character and Boat representation
    plainText = '\033[2m'+"❖"+'\x1b[0m'+" {} "+'\x1b[4;35;43m'+"|"+'\x1b[0m'+'\x1b[1;33;34m' + \
        "~~~~~~~~~~~~~~~~~"+'\x1b[0m'+'\x1b[4;35;43m' + \
        "|" + '\x1b[0m'+" {} "+('\033[2m'+"❖"+'\x1b[0m')

    #Displaying the graphical representation of the state
    print(('\033[2m'+"✵"+'\x1b[0m') * 44)
    print(plainText.format(shore, shore))

    for i in range(0, 8):
        characterName = listOfChars[i] + \
            ('\x1b[1;32;42m'+"⥮"+'\x1b[0m') * (10 - len(listOfChars[i]))
        if state[i] == 0:
            print(plainText.format(
                '\x1b[7;35;46m'+characterName+'\x1b[0m', shore))
        else:
            print(plainText.format(
                shore, '\x1b[7;35;46m'+characterName+'\x1b[0m'))
        if i == 3:
            # Displaying boat on the right side of the river
            if state[8]:
                print('\033[2m'+"❖"+'\x1b[0m', shore, '\x1b[4;35;43m'+"|"+'\x1b[0m', '\x1b[1;33;34m'+"~~~~~~~~~~~" +
                      '\x1b[0m', '\x1b[1;34;41m'+"⛵"+'\x1b[0m', '\x1b[4;35;43m'+"|"+'\x1b[0m', shore, '\033[2m'+"❖"+'\x1b[0m')
            else:
                #Displaying boat on the left side of the river
                print('\033[2m'+"❖"+'\x1b[0m', shore, '\x1b[4;35;43m'+"|"+'\x1b[0m', '\x1b[1;34;41m'+"⛵"+'\x1b[0m',
                      '\x1b[1;33;34m'+"~~~~~~~~~~~"+'\x1b[0m', '\x1b[4;35;43m'+"|"+'\x1b[0m', shore, '\033[2m'+"❖"+'\x1b[0m')

    print(plainText.format(shore, shore))
    print(('\033[2m'+"✵"+'\x1b[0m') * 44)

# ShowPath:  Shows Multiple States in order
def ShowPath(List_States):
    # Check if there are any state to show
    if List_States == None:
```

```python
        print("\x1B[41;2;35mThere is Nothing To Show\033[0m")
        exit(-1)
    counter = 1
    previousState = List_States[0]
    # Displaying each state in the list of states with a delay
    for state in List_States[1:]:
        if counter != 1:
            sleep(0.3)
        Clear()
        print(f"\033[3;46;35mChild State {counter}\033[0m")
        Show(state)
        TellMove(previousState, state)
        counter += 1
        previousState = state

# Clear: Clears The Terminal output
def Clear():
    if name == 'nt':
        system('cls')
    else:
        system('Clear')

# Assigning values to problem members
POLICE = 0; THIEF = 1; FATHER = 2; MOTHER = 3; DAUGHTER_1 = 4; DAUGHTER_2 = 5; SON_1 = 6; SON_2 = 7;
BOAT_Direction = 8

# Checks whether a state is valid
def IsValid(state):
            # checking conflicts with Daughters
    return ((state[DAUGHTER_1] == state[MOTHER] or state[DAUGHTER_1] != state[FATHER]) and (
            state[DAUGHTER_2] == state[MOTHER] or state[DAUGHTER_2] != state[FATHER])) and ((

            # checking conflicts with Sons
            state[SON_1] == state[FATHER] or state[SON_1] != state[MOTHER]) and (
            state[SON_2] == state[FATHER] or state[SON_2] != state[MOTHER])) and (

            # checking conflicts with Thief
            state[POLICE] == state[THIEF] or (state[THIEF] != state[FATHER] and
            state[THIEF] != state[MOTHER] and state[THIEF] != state[DAUGHTER_1] and
            state[THIEF] != state[DAUGHTER_2] and state[THIEF] != state[SON_1] and
            state[THIEF] != state[SON_2]))

# checks if the state is the Goal
def IsGoal(state):
    return state == [1, 1, 1, 1, 1, 1, 1, 1, 1]

# it generates all states from a valid state and filters all invalid ones
def GenerateAllValidStates(state):

    if not IsValid(state):
        print('\n'+"\x1B[41;1;35mSorry I can\'t Generate States for an Invalid State\033[0m")
        exit(-1)
    # creating a new empty list of valid states
    validStates = []
```

```python
    # for each character in the below list , do:
    for currentCharacter in [POLICE, THIEF, FATHER, MOTHER, DAUGHTER_1, DAUGHTER_2, SON_1, SON_2]:
        #for each parent in the below list do:
        for parent in [FATHER, MOTHER, POLICE]:
            # if all three of currentCharacter and parent and boat_Direction Directions were on the same side of the river:
            if state[currentCharacter] == state[parent] == state[BOAT_Direction]:

                # making a shallow copy of the state and assiging it to the new variable
                new_State = copy(state)

                if new_State[currentCharacter]: # making a new state by moving all those characters to right side of the river
                    new_State[currentCharacter] = new_State[parent] = new_State[BOAT_Direction] = 0
                else:                        # making a new state by moving all those characters to left side of the river
                    new_State[currentCharacter] = new_State[parent] = new_State[BOAT_Direction] = 1
                # Checks whether the state is valid and not duplicated
                if IsValid(new_State) and new_State not in validStates:
                    validStates.append(new_State)

    return validStates

# Describes State Changes in Context
def TellMove(state, new_state):
    peopleList = ['POLICE', 'THIEF', 'FATHER', 'MOTHER', 'DAUGHTER_1', 'DAUGHTER_2', 'SON_1', 'SON_2',
'BOAT_Direction']

    diff = list()
    for item1, item2 in zip(state, new_state):
        item = item1 - item2
        diff.append(item)

    Direction = 'RIGHT' if diff[8] == -1 else 'LEFT'

    movedPeople = list()
    for i in range(8):
        if diff[i] == -1 or diff[i] == 1:
            movedPeople.append(i)
    if len(movedPeople) == 1:

        print("\n" + f"\033[4;43;35m{peopleList[movedPeople[0]]}\033[0m" +
              ' moved to the ' f"\033[3;44;30m{Direction}\033[0m")
    else:
        print("\n" + f"\033[4;43;35m{peopleList[movedPeople[0]]}\033[0m"' and ' +
              f"\033[4;43;35m{peopleList[movedPeople[1]]}\033[0m"+' moved to the ' f"\033[3;44;30m{Direction}\033[0m")

# Non Recursive Depth-Limited-Search with viewAllStatesFlag feature
def Iterative_DLS(state, DEPTH_LIMIT, viewAllStatesFlag):
    # if state is not valid then exit()
    if not IsValid(state):
        print(
            '\n'+"\x1B[41;1;35mSorry I cannot Find a Soution for an Invalid State\033[0m")
        exit(-1)

    #tuple of currentState, currentStateDepth
    unExpandedNodes = [(state, 0)]
```

```python
    expandedNodesList = []

    # while unExpandedNodes list is not empty , do the below code segment
    while unExpandedNodes is not None:
        # Pops up the last node from unExpanded nodes list, and return state and depth of that node
        # into two seperate variables
        stateOfLastUnexpandedNode,depthOfLastUnexpandedNode= unExpandedNodes.pop()

        # checks if current poped up state is goal or not
        if IsGoal(stateOfLastUnexpandedNode):
            expandedNodesList.append((stateOfLastUnexpandedNode, depthOfLastUnexpandedNode))
            # viewAllStatesFlag status, allow us to choose whether to show all ordered States or not
            if viewAllStatesFlag:
                allCheckedStatesList = []
                for eachNode in expandedNodesList:
                    allCheckedStatesList.append(eachNode[0])
                return allCheckedStatesList
            else:
                # if we dont want to view all states, we have to return just the true answer state
                trueAnswerStatesList = FilterFinalAnswerStates(expandedNodesList)
                return trueAnswerStatesList

        # checks wheter current depth Reached to the DEPTH_LIMIT
        if depthOfLastUnexpandedNode < DEPTH_LIMIT:
            # initializing a empty list for expanded States of Nodes
            expandedStatesList = []
            for i in expandedNodesList:
                expandedStatesList.append(i[0])

            # the current unexpanded node doesn't exist in expandedStates List, then do
            if stateOfLastUnexpandedNode not in expandedStatesList:

                # adds the current tuple of state and depth to expanded Nodes list
                expandedNodesList.append((stateOfLastUnexpandedNode, depthOfLastUnexpandedNode))
                # then it Generates all children of that state
                generatedChildrenNodesList = GenerateAllValidStates(stateOfLastUnexpandedNode)
                # increasing the depth of children nodes, by one
                depthOfLastUnexpandedNode += 1
                depthOfChlidrenNodes = [depthOfLastUnexpandedNode]*(len(generatedChildrenNodesList))
                # creating a pair of children states and there depths and assigning it to the new tuple variable
                tupleOfChildrenAndDepthsList = tuple(zip(generatedChildrenNodesList, depthOfChlidrenNodes))
                # Adding a new children Nodes list to the unExpanded Nodes list
                unExpandedNodes.extend(tupleOfChildrenAndDepthsList)

# Filter Final True Answer States from the list of states
def FilterFinalAnswerStates(finalNodes):
    lastDepth = (finalNodes[-1])[1]
    finalNodes.reverse()
    filteredStates = []
    # the node is tuple of currentState, currentStateDepth
    # combination of for and if expressions to select the first most innerDepth nodes
    # this segment of code finds true parents and ancestors for the Goal Node
    for node in finalNodes:
```

```python
        if node[1] < lastDepth:
            filteredStates.append(node[0])  #appends state
            lastDepth = node[1]             #changes lastDepth value with currentStateDepth
    # at the end we have to reverse the result list and appending the goal node to the end of it
    filteredStates.reverse()
    filteredStates.append((finalNodes[0])[0])
    return filteredStates

# main part of the Code, Calling Iterative_DLS on begin state=[0..0]
# ///////////////MAIN////////////////

viewAllStatesFlag = False
startState = [0]*9
# calling DLS on initial State
finalStates = Iterative_DLS(startState, 20, viewAllStatesFlag)
# priniting searched result using ShowPath function in terminal
ShowPath(finalStates)
# By Parsa Yousefi Nejad
```