**Experiment NO: 1**  **Date:**

**Title:** Missionaries and Cannibals Problem

**Aim:** To implement solutions to the missionary-cannibals problem using breadth-first search technique.

**Description:**Missionaries and cannibals is a classic river-crossing puzzle, popular as a toy problem in artificial intelligence. In this problem, three missionaries and three cannibals must cross a river using a boat that carries at most two persons. This must be done under the constraint that on either side of the riverbank, the number of cannibals must not outnumber the number of missionaries. This is because if this case occurs, the cannibals on the dominant side will eat the missionaries. The boat cannot cross the river by itself.  
To formulate a solution to the problem, firstly we need to formulate the problem in a manner where understanding the solution steps become easier. The accepted convention is to use a 3-tuple to represents approachable states: (m,c,b) ,where m stands for number of missionaries on the left bank, c stands for number of cannibals on left bank, and b stands for the side the boat is on (in that state)[usually, we use left/right, or 0 for left side and 1 for right side]. We assume the start state is (3,3,0), because there are three each missionary and cannibals. We create a state space tree , by representing (3,3,0) as the root of the tree, and with children of each node representing the state that can be accessed via that state(viz, by a movement of 1 or 2 persons and switch in boat side), such that the constraints of the problem are not violated. To prevent infinite loops, we decide to keep track of which steps have been already obtained (maybe through some alternate path), and never append a child state that has already been visited/obtained.  
We use Breadth-First Search technique to solve this problem. In this technique, we use FIFO(First In First Out). Thus at every step, we select a node(state) from the state-space tree, check if it is the final state, and if not, generate all the VALID states that can be approached from Chosen state and that have not already been visited. In this manner, the steps continue, as long as there is one valid state left to be checked. For implementing the program for this problem, we use the data structure **queue** , which follows the first-in-first-out approach.

**Program:**

from collections import deque

import time,sys

# missionaries and cannibals problem

# solution using breadth first search

def equal\_tuple(tup1,tup2,n):

for i in range(n):

if tup1[i] != tup2[i]:

return False

return True

"""

the state tuple is (m,c,b)

m is no of missionaries on left side of river,

c is no of cannibals on left side of river

b is a boolean representing boat: if true, on the left:1, if false, on the right:0

"""

def is\_valid(initialState,currentState):

"""

initial state is a 2-tuple that tells the initial state of the system,

current state is a 3-tuple that tells the current state of the system

in no way can m2 be less than c2,

and also, we cannot have (m1-m2)<(c1-c2)

"""

m1 = initialState[0]

m2 = currentState[0]

c1 = initialState[1]

c2 = currentState[1]

if (m2<c2) and m2!=0:

# if missionaries on one side are less than cannibals, as long as there ar no zero missionaries

return False

#w: other side, check

if m2!=0 and (c1-c2)>(m1-m2) and (m1-m2 != 0) and m2!=0:

# if missionaries on other side are less than cannibals

return False

return True

i = 0

visited = {():()} # w: where each child stores its parent

# if valid solution found, store, else, remove that sublist

def successorStates(initialState,givenState):

# w: here givenState is the current

# do the addition thing, viz,

# subtract two cannibals

# subtract two missionaries,

# 1 cannibal 1 missionaries

# subtract 1 cannibal

#w: givenstate[2] is 0 when on left, 1 when on right

st = [] # contains the successful solution lists, and therefore, is supposed to be a list of lists

ml = givenState[0] # no of missionaries on the left right now

cl = givenState[1] # no of cannibals on the left right now

m2 = initialState[0] # no of missionaries on the left at the start

c2 = initialState[1] # no of cannibals of the right at the start

boat = 1-givenState[2] # w: we are changing the side at the start itself

states = []

if boat == 1:

print("we are going to the right")

# todo : WE NEED TO SEND ONE OR MORE PEOPLE ON THE RIGHT TO THE LEFT

#LG: so we have to add to the tuple because no on left MUST INCREASE

extraStates = []

try:

for i in range(0,m2):

for j in range(0,c2):

if (i+j)>0 and (i+j)<=2:

# print(list(visited.values()))

if (ml-i,cl-j,boat) not in (list(visited.values())+list(visited.keys()) ):

# print(f"({ml-i},{cl-j},{boat}) is not in ({list(visited.values())})",end="\t")

if (ml-i)>=0 and (cl-j)>=0:

if (ml==i) and (cl==j):

# w: we have reached the success state

# w: have to point the success to parent

print("Reached success")

visited[(0,0,1)] = givenState

if is\_valid(initialState,(ml-i,cl-j,boat)):

extraStates.append((ml-i, cl-j,boat))

if m2==ml and c2==cl:

#w: initial state

s = (0,0,0)

visited[s] = (0,0,0)

except Exception as e:

print(f"cMiss = {m2-i}, cCann = {c2-j}")

# print(f"new states valid: {extraStates}")

# print("right")

for state in extraStates:

if is\_valid((m2,c2,0),state):

visited[state] = givenState

states.append(state) #W: where states is a queue of tasks

elif boat == 0:

print("we are going to the left")

extraStates = []

try:

for i in range(0,m2):

for j in range(0,c2):

# print(f"\ny1 , i={i},j={j} ",end=" ")

if (i+j)>0 and (i+j)<=2: #w: ensures max only two ppl travel

# print(f"y2, i={i},j={j} ",end=" ")

if (ml+i,cl+j,boat) not in (list(visited.values())+list(visited.keys()) ):

# print(f"({ml+i},{cl+j},{boat}) is not in {list(visited.values())}",end="\t")

# print(f"y3, i={i},j={j} ",end=" ")

if (ml+i)<=m2 and (cl+j)<=c2:

# print(f"\tml+i = {ml+i},cl+j={cl+j},boatGoing={boat}",end=" ")

if is\_valid(initialState,(ml+i,cl+j,boat)):

# print(f"y5, i={i},j={j} (adding new state = ({ml+i},{cl+j},{boat}))",end="")

extraStates.append((ml+i, cl+j,boat))

except IndexError as e:

print(f"cMiss = {givenState[0]}, cCann = {givenState[1]}")

# print(f"New states are {extraStates}")

# time.sleep(2)

for state in extraStates:

if is\_valid((m2,c2,0),state):

visited[state] = givenState #w: setting it to parent

states.append(state)

return states

def mnc(initialState):

"""

takes a 3 item list as input,

first is missionaries, second is cannibals, third is boat(0 on left, 1 on right)

"""

# initialise a queue

q = deque()

q.append(initialState)

# ! assuming initial state is a 3-item tuple

# now the bfs loop

j = 0

while q:

# print(visited.values())

c = q.popleft()

print(f"current= {c}")

j += 1

if equal\_tuple(c,(0,0,1),3):

print(f"found a solution: {c} ")

solutionQ = list()

x = (0,0,0)

s = successorStates(initialState,c)

solutionQ.append((0,0,1))

t = (0,0,1)

while x is not initialState:

x = visited[t]

solutionQ.append(x)

t = x

print("Solution Q:")

for i in solutionQ[::-1]:

print(f"=>{i} ",end="")

# generate the successive states

else:

successor = successorStates(initialState,c)

if not successor:

print("No new states, kill this branch")

else:

print(f"New states are {successor}")

for state in successor:

#w: have given the validity check job to the successorStates function

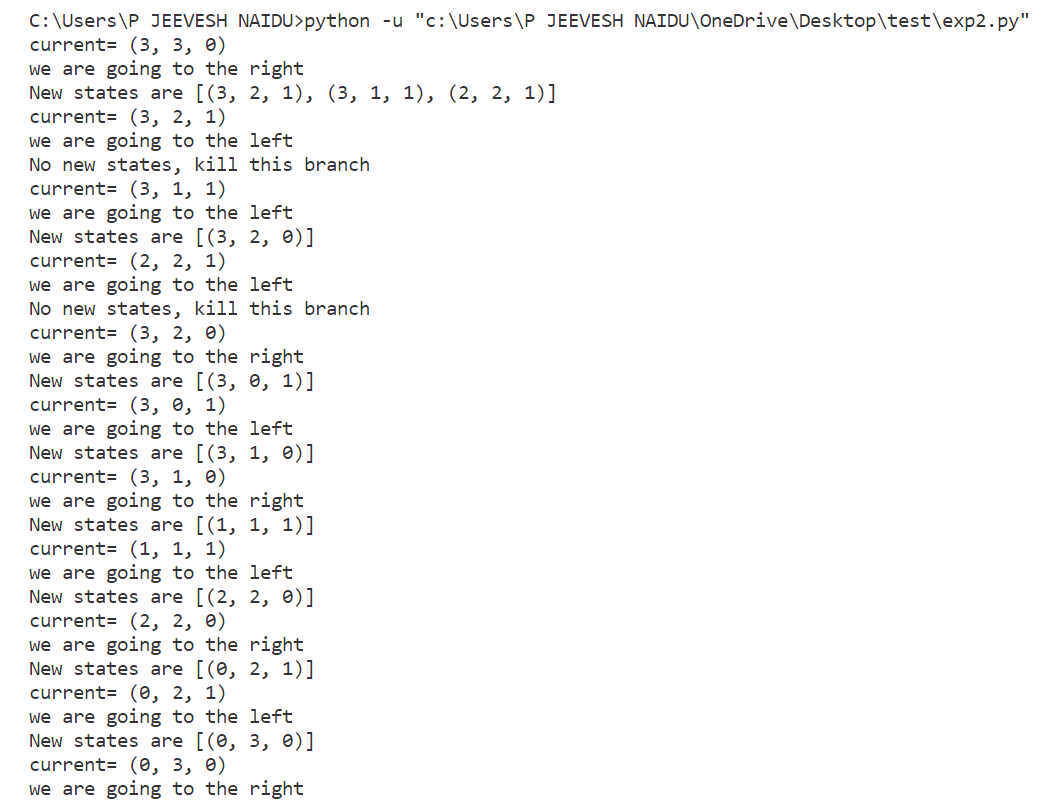
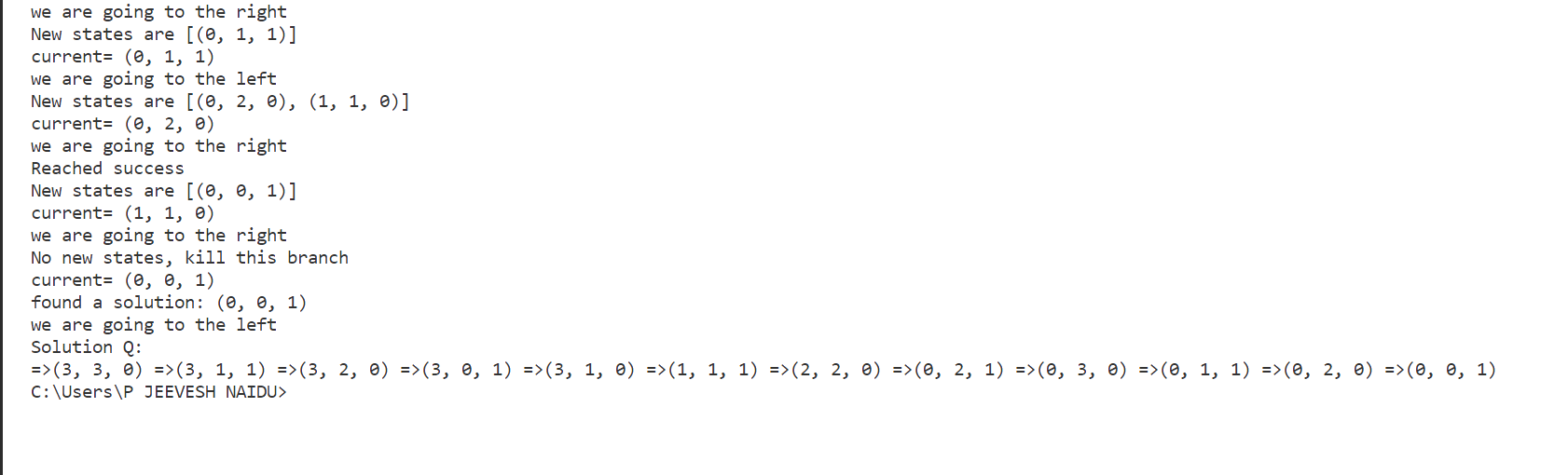
q.append(state)

# print(q)

# w: should be a 4 tuple, so that we can count which indice also

mnc((3,3,0))

**Output:**

**Conclusion:**

Studied the missionaries and cannibals’ problem and implemented its solution using breadth-first search technique in Python.