**Toy Problem - Tic Toe Problem:**

import os

import time

board = [' ',' ',' ',' ',' ',' ',' ',' ',' ',' ']

player = 1

########win Flags##########

Win = 1

Draw = -1

Running = 0

Stop = 1

###########################

Game = Running

Mark = 'X'

#This Function Draws Game Board

def DrawBoard():

print(" %c | %c | %c " % (board[1],board[2],board[3]))

print("\_\_\_|\_\_\_|\_\_\_")

print(" %c | %c | %c " % (board[4],board[5],board[6]))

print("\_\_\_|\_\_\_|\_\_\_")

print(" %c | %c | %c " % (board[7],board[8],board[9]))

print(" | | ")

#This Function Checks position is empty or not

def CheckPosition(x):

if(board[x] == ' '):

return True

else:

return False

#This Function Checks player has won or not

def CheckWin():

global Game

#Horizontal winning condition

if(board[1] == board[2] and board[2] == board[3] and board[1] != ' '):

Game = Win

elif(board[4] == board[5] and board[5] == board[6] and board[4] != ' '):

Game = Win

elif(board[7] == board[8] and board[8] == board[9] and board[7] != ' '):

Game = Win

#Vertical Winning Condition

elif(board[1] == board[4] and board[4] == board[7] and board[1] != ' '):

Game = Win

elif(board[2] == board[5] and board[5] == board[8] and board[2] != ' '):

Game = Win

elif(board[3] == board[6] and board[6] == board[9] and board[3] != ' '):

Game=Win

#Diagonal Winning Condition

elif(board[1] == board[5] and board[5] == board[9] and board[5] != ' '):

Game = Win

elif(board[3] == board[5] and board[5] == board[7] and board[5] != ' '):

Game=Win

#Match Tie or Draw Condition

elif(board[1]!=' ' and board[2]!=' ' and board[3]!=' ' and board[4]!=' ' and board[5]!=' ' and board[6]!=' ' and board[7]!=' ' and board[8]!=' ' and board[9]!=' '):

Game=Draw

else:

Game=Running

print("Tic-Tac-Toe Game Designed By Sourabh Somani")

print("Player 1 [X] --- Player 2 [O]\n")

print()

print()

print("Please Wait...")

time.sleep(3)

while(Game == Running):

os.system('cls')

DrawBoard()

if(player % 2 != 0):

print("Player 1's chance")

Mark = 'X'

else:

print("Player 2's chance")

Mark = 'O'

choice = int(input("Enter the position between [1-9] where you want to mark : "))

if(CheckPosition(choice)):

board[choice] = Mark

player+=1

CheckWin()

os.system('cls')

DrawBoard()

if(Game==Draw):

print("Game Draw")

elif(Game==Win):

player-=1

if(player%2!=0):

print("Player 1 Won")

else:

print("Player 2 Won"**)**

**Vacuum Cleaner problem:**

#INSTRUCTIONS

#Enter LOCATION A/B in captial letters

#Enter Status O/1 accordingly where 0 means CLEAN and 1 means DIRTY

def vacuum\_world():

# initializing goal\_state

# 0 indicates Clean and 1 indicates Dirty

goal\_state = {'A': '0', 'B': '0'}

cost = 0

location\_input = input("Enter Location of Vacuum") #user\_input of location vacuum is placed

status\_input = input("Enter status of " + location\_input) #user\_input if location is dirty or clean

status\_input\_complement = input("Enter status of other room")

print("Initial Location Condition" + str(goal\_state))

if location\_input == 'A':

# Location A is Dirty.

print("Vacuum is placed in Location A")

if status\_input == '1':

print("Location A is Dirty.")

# suck the dirt and mark it as clean

goal\_state['A'] = '0'

cost += 1 #cost for suck

print("Cost for CLEANING A " + str(cost))

print("Location A has been Cleaned.")

if status\_input\_complement == '1':

# if B is Dirty

print("Location B is Dirty.")

print("Moving right to the Location B. ")

cost += 1 #cost for moving right

print("COST for moving RIGHT" + str(cost))

# suck the dirt and mark it as clean

goal\_state['B'] = '0'

cost += 1 #cost for suck

print("COST for SUCK " + str(cost))

print("Location B has been Cleaned. ")

else:

print("No action" + str(cost))

# suck and mark clean

print("Location B is already clean.")

if status\_input == '0':

print("Location A is already clean ")

if status\_input\_complement == '1':# if B is Dirty

print("Location B is Dirty.")

print("Moving RIGHT to the Location B. ")

cost += 1 #cost for moving right

print("COST for moving RIGHT " + str(cost))

# suck the dirt and mark it as clean

goal\_state['B'] = '0'

cost += 1 #cost for suck

print("Cost for SUCK" + str(cost))

print("Location B has been Cleaned. ")

else:

print("No action " + str(cost))

print(cost)

# suck and mark clean

print("Location B is already clean.")

else:

print("Vacuum is placed in location B")

# Location B is Dirty.

if status\_input == '1':

print("Location B is Dirty.")

# suck the dirt and mark it as clean

goal\_state['B'] = '0'

cost += 1 # cost for suck

print("COST for CLEANING " + str(cost))

print("Location B has been Cleaned.")

if status\_input\_complement == '1':

# if A is Dirty

print("Location A is Dirty.")

print("Moving LEFT to the Location A. ")

cost += 1 # cost for moving right

print("COST for moving LEFT" + str(cost))

# suck the dirt and mark it as clean

goal\_state['A'] = '0'

cost += 1 # cost for suck

print("COST for SUCK " + str(cost))

print("Location A has been Cleaned.")

else:

print(cost)

# suck and mark clean

print("Location B is already clean.")

if status\_input\_complement == '1': # if A is Dirty

print("Location A is Dirty.")

print("Moving LEFT to the Location A. ")

cost += 1 # cost for moving right

print("COST for moving LEFT " + str(cost))

# suck the dirt and mark it as clean

goal\_state['A'] = '0'

cost += 1 # cost for suck

print("Cost for SUCK " + str(cost))

print("Location A has been Cleaned. ")

else:

print("No action " + str(cost))

# suck and mark clean

print("Location A is already clean.")

# done cleaning

print("GOAL STATE: ")

print(goal\_state)

print("Performance Measurement: " + str(cost))

vacuum\_world()

**Missionaries and Cannibals Problem:**

The missionaries and cannibals problem, which is a famous problem in AI, is usually stated as follows. 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.

import math

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Missionaries and Cannibals Problem

class State():

def \_\_init\_\_(self, cannibalLeft, missionaryLeft, boat, cannibalRight, missionaryRight):

self.cannibalLeft = cannibalLeft

self.missionaryLeft = missionaryLeft

self.boat = boat

self.cannibalRight = cannibalRight

self.missionaryRight = missionaryRight

self.parent = None

def is\_goal(self):

if self.cannibalLeft == 0 and self.missionaryLeft == 0:

return True

else:

return False

def is\_valid(self):

if self.missionaryLeft >= 0 and self.missionaryRight >= 0 \

and self.cannibalLeft >= 0 and self.cannibalRight >= 0 \

and (self.missionaryLeft == 0 or self.missionaryLeft >= self.cannibalLeft) \

and (self.missionaryRight == 0 or self.missionaryRight >= self.cannibalRight):

return True

else:

return False

def \_\_eq\_\_(self, other):

return self.cannibalLeft == other.cannibalLeft and self.missionaryLeft == other.missionaryLeft \

and self.boat == other.boat and self.cannibalRight == other.cannibalRight \

and self.missionaryRight == other.missionaryRight

def \_\_hash\_\_(self):

return hash((self.cannibalLeft, self.missionaryLeft, self.boat, self.cannibalRight, self.missionaryRight))

def successors(cur\_state):

children = [];

if cur\_state.boat == 'left':

new\_state = State(cur\_state.cannibalLeft, cur\_state.missionaryLeft - 2, 'right',

cur\_state.cannibalRight, cur\_state.missionaryRight + 2)

## Two missionaries cross left to right.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

new\_state = State(cur\_state.cannibalLeft - 2, cur\_state.missionaryLeft, 'right',

cur\_state.cannibalRight + 2, cur\_state.missionaryRight)

## Two cannibals cross left to right.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

new\_state = State(cur\_state.cannibalLeft - 1, cur\_state.missionaryLeft - 1, 'right',

cur\_state.cannibalRight + 1, cur\_state.missionaryRight + 1)

## One missionary and one cannibal cross left to right.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

new\_state = State(cur\_state.cannibalLeft, cur\_state.missionaryLeft - 1, 'right',

cur\_state.cannibalRight, cur\_state.missionaryRight + 1)

## One missionary crosses left to right.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

new\_state = State(cur\_state.cannibalLeft - 1, cur\_state.missionaryLeft, 'right',

cur\_state.cannibalRight + 1, cur\_state.missionaryRight)

## One cannibal crosses left to right.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

else:

new\_state = State(cur\_state.cannibalLeft, cur\_state.missionaryLeft + 2, 'left',

cur\_state.cannibalRight, cur\_state.missionaryRight - 2)

## Two missionaries cross right to left.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

new\_state = State(cur\_state.cannibalLeft + 2, cur\_state.missionaryLeft, 'left',

cur\_state.cannibalRight - 2, cur\_state.missionaryRight)

## Two cannibals cross right to left.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

new\_state = State(cur\_state.cannibalLeft + 1, cur\_state.missionaryLeft + 1, 'left',

cur\_state.cannibalRight - 1, cur\_state.missionaryRight - 1)

## One missionary and one cannibal cross right to left.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

new\_state = State(cur\_state.cannibalLeft, cur\_state.missionaryLeft + 1, 'left',

cur\_state.cannibalRight, cur\_state.missionaryRight - 1)

## One missionary crosses right to left.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

new\_state = State(cur\_state.cannibalLeft + 1, cur\_state.missionaryLeft, 'left',

cur\_state.cannibalRight - 1, cur\_state.missionaryRight)

## One cannibal crosses right to left.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

return children

def breadth\_first\_search():

initial\_state = State(3,3,'left',0,0)

if initial\_state.is\_goal():

return initial\_state

frontier = list()

explored = set()

frontier.append(initial\_state)

while frontier:

state = frontier.pop(0)

if state.is\_goal():

return state

explored.add(state)

children = successors(state)

for child in children:

if (child not in explored) or (child not in frontier):

frontier.append(child)

return None

def print\_solution(solution):

path = []

path.append(solution)

parent = solution.parent

while parent:

path.append(parent)

parent = parent.parent

for t in range(len(path)):

state = path[len(path) - t - 1]

print "(" + str(state.cannibalLeft) + "," + str(state.missionaryLeft) \

+ "," + state.boat + "," + str(state.cannibalRight) + "," + \

str(state.missionaryRight) + ")"

def main():

solution = breadth\_first\_search()

print "Missionaries and Cannibals solution:"

print "(cannibalLeft,missionaryLeft,boat,cannibalRight,missionaryRight)"

print\_solution(solution)

# if called from the command line, call main()

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Water Jug Problem**

# This function is used to initialize the

# dictionary elements with a default value.

from collections import defaultdict

# jug1 and jug2 contain the value

# for max capacity in respective jugs

# and aim is the amount of water to be measured.

jug1, jug2, aim = 4, 3, 2

# Initialize dictionary with

# default value as false.

visited = defaultdict(lambda: False)

# Recursive function which prints the

# intermediate steps to reach the final

# solution and return boolean value

# (True if solution is possible, otherwise False).

# amt1 and amt2 are the amount of water present

# in both jugs at a certain point of time.

def waterJugSolver(amt1, amt2):

# Checks for our goal and

# returns true if achieved.

if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):

print(amt1, amt2)

return True

# Checks if we have already visited the

# combination or not. If not, then it proceeds further.

if visited[(amt1, amt2)] == False:

print(amt1, amt2)

# Changes the boolean value of

# the combination as it is visited.

visited[(amt1, amt2)] = True

# Check for all the 6 possibilities and

# see if a solution is found in any one of them.

return (waterJugSolver(0, amt2) or

waterJugSolver(amt1, 0) or

waterJugSolver(jug1, amt2) or

waterJugSolver(amt1, jug2) or

waterJugSolver(amt1 + min(amt2, (jug1-amt1)),

amt2 - min(amt2, (jug1-amt1))) or

waterJugSolver(amt1 - min(amt1, (jug2-amt2)),

amt2 + min(amt1, (jug2-amt2))))

# Return False if the combination is

# already visited to avoid repetition otherwise

# recursion will enter an infinite loop.

else:

return False

print("Steps: ")

# Call the function and pass the

# initial amount of water present in both jugs.

waterJugSolver(0, 0)

**A\* search Algorithm**