**EXPERIMENT 1 – TOY PROBLEM (WATER JUG)**

**ALGORITHM and CODE**

/\*(X,Y | X<4)

# (4,Y)

# {Fill 4-gallon jug}

# (X,Y |Y<3)

# (X,3)

# {Fill 3-gallon jug}

# (X,Y |X>0)

# (0,Y)

# {Empty 4-gallon jug}

# (X,Y | Y>0)

# (X,0)

# {Empty 3-gallon jug}

# (X,Y | X+Y>=4 ^ Y>0)

# (4,Y-(4-X))

# {Pour water from 3-gallon jug into 4-gallon jug until 4-gallon jug is full}

# (X,Y | X+Y>=3

# ^X>0)

# (X-(3-Y),3)

# {Pour water from 4-gallon jug into 3-gallon jug until 3-gallon jug is full}

# (X,Y | X+Y<=4

# ^Y>0)

# (X+Y,0)

# {Pour all water from 3-gallon jug into 4-gallon jug}

# (X,Y | X+Y <=3^ X>0)

# (0,X+Y)

# {Pour all water from 4-gallon jug into 3-gallon jug}

# (0,2)

# (2,0)

# {Pour 2 gallon water from 3 gallon jug into 4 gallon jug}

x=int(input("Enter X : "))

y=int(input("Enter Y : "))

while True:

rn=int(input("Enter The Rule No. : "))

if rn==2:

if y<3:

x=0

y=3

if rn==3:

if x>0:

x=0

y=3

if rn==5:

if x+y>4:

x=4

y=y-(4-x)

if rn==7:

if x+y<4:

x=x+y

y=0

if rn==9:

x=2

y=0

print("X =",x)

print("Y =",y)

if x==2:

print("The result is a goal state.")

break

**EXPERIMENT 2 – REAL WORLD AGENT (VACCUM)**

import random

def display(room):

print(room)

room = [

[1, 1, 1, 1],

[1, 1, 1, 1],

[1, 1, 1, 1],

[1, 1, 1, 1],

]

print("All the rooom are dirty")

display(room)

x =0

y= 0

while x < 4:

while y < 4:

room[x][y] = random.choice([0,1])

y+=1

x+=1

y=0

print("Before cleaning the room I detect all of these random dirts")

display(room)

x =0

y= 0

z=0

while x < 4:

while y < 4:

if room[x][y] == 1:

print("Vaccum in this location now,",x, y)

room[x][y] = 0

print("cleaned", x, y)

z+=1

y+=1

x+=1

y=0

pro= (100-((z/16)\*100))

print("Room is clean now!")

display(room)

print('Performance =',pro,'%')

**EXPERIMENT 3 – CONSTRAINT SATISFACTORY (N-QUEEN)**

# while there are untried configurations

# {

# generate the next configuration

# if queens don't attack in this configuration then

# {

# print this configuration;

# }

# }

class queen():

N = 4

vd = [0] \* 30

rs = [0] \* 30

ad = [0] \* 30

def nqueen(self):

board = [[0, 0, 0, 0],

[0, 0, 0, 0],

[0, 0, 0, 0],

[0, 0, 0, 0], ]

if (self.solve(board, 0) == False):

print('There is no solution of the problem !!!')

return False

self.printSol(board)

return True

def solve(self, board, col):

if (col >= self.N):

return True

for i in range(self.N):

if ((self.vd[i - col + self.N - 1] != 1 and self.rs[i + col] != 1) and self.ad[i] != 1):

board[i][col] = 1

self.vd[i - col + self.N -

1] = self.rs[i + col] = self.ad[i] = 1

if (self.solve(board, col + 1)):

return True

board[i][col] = 0

self.vd[i - col + self.N -

1] = self.rs[i + col] = self.ad[i] = 0

return False

def printSol(self, board):

print("The solution to the board is : ")

for i in range(self.N):

for j in range(self.N):

print(board[i][j], end=' ')

print()

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

queen().nqueen()

**EXPERIMENT 4 - BFS AND DFS**

# Create a node list (Queue) that initially contains the first node N and mark it as visited.

# Visit the adjacent unvisited vertex of N and insert it in a queue.

# If there are no remaining adjacent vertices left, remove the first vertex from the queue mark it as visited, display it.

# Repeat step 1 and step 2 until the queue is empty or the desired node is found.

graph = {

'S': ['A', 'B'],

'A': ['C', 'D'],

'B': ['G', 'H'],

'C': ['E', 'F'], 'D': [],

'G': ['I'],

'H': [],

'E': ['K'],

'F': [],

'I': [],

'K': []

}

visited = []

queue = []

def bfs(visited,graph,node):

visited.append(node)

queue.append(node)

while queue:

P=queue.pop(0)

print(P,end=" ")

for neighbour in graph[P]:

if neighbour not in visited:

visited.append(neighbour)

queue.append(neighbour)

avisit = set()

def dfs(avisit,graph,node):

if node not in avisit:

print(node,end=" ")

avisit.add(node)

for neighbour in graph[node]:

dfs(avisit,graph,neighbour)

print("Breadth first search")

bfs(visited,graph,'S')

print("\nDepth first search")

dfs(avisit,graph,'S')

**EXPERIMENT 5 – BEST FIRST AND A\* SEARCH ALGORITHM**

**BEST FIRST SEARCH:**

# Place the starting node into the OPEN list.

# If the OPEN list is empty, Stop and return failure.

# Remove the node n, from the OPEN list which has the lowest value of h(n), and places it in the CLOSED list.

# n is goal, then return else

# Expand the node n, and generate and check the successors of node n. and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 5.

# For each successor node, algorithm checks for evaluation function f(n), and then check if the node has been in either OPEN or CLOSED list. If the node has not been in both lists, then add it to the OPEN list.

# Return to Step 2.

class Graph:

def \_\_init\_\_(self, graph\_dict=None, directed=True):

self.graph\_dict = graph\_dict or {}

self.directed = directed

if not directed:

self.make\_undirected()

def make\_undirected(self):

for a in list(self.graph\_dict.keys()):

for (b, dist) in self.graph\_dict[a].items():

self.graph\_dict.setdefault(b, {})[a] = dist

def connect(self, A, B, distance=1):

self.graph\_dict.setdefault(A, {})[B] = distance

if not self.directed:

self.graph\_dict.setdefault(B, {})[A] = distance

def get(self, a, b=None):

links = self.graph\_dict.setdefault(a, {})

if b is None:

return links

else:

return links.get(b)

def nodes(self):

s1 = set([k for k in self.graph\_dict.keys()])

s2 = set([k2 for v in self.graph\_dict.values()

for k2, v2 in v.items()])

nodes = s1.union(s2)

return list(nodes)

class Node:

def \_\_init\_\_(self, name: str, parent: str):

self.name = name

self.parent = parent

self.g = 0

self.h = 0

self.f = 0

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

return self.name == other.name

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

return self.f < other.f

def \_\_repr\_\_(self):

return ('({0},{1})'.format(self.position, self.f))

def best\_first\_search(graph, heuristics, start, end):

open = []

closed = []

start\_node = Node(start, None)

goal\_node = Node(end, None)

open.append(start\_node)

while len(open) > 0:

open.sort()

current\_node = open.pop(0)

closed.append(current\_node)

if current\_node == goal\_node:

path = []

while current\_node != start\_node:

path.append(current\_node.name + ': ' + str(current\_node.g))

current\_node = current\_node.parent

path.append(start\_node.name + ': ' + str(start\_node.g))

return path[::-1]

neighbors = graph.get(current\_node.name)

for key, value in neighbors.items():

neighbor = Node(key, current\_node)

if(neighbor in closed):

continue

neighbor.g = current\_node.g + \

graph.get(current\_node.name, neighbor.name)

neighbor.h = heuristics.get(neighbor.name)

neighbor.f = neighbor.h

if(add\_to\_open(open, neighbor) == True):

open.append(neighbor)

return None

def add\_to\_open(open, neighbor):

for node in open:

if (neighbor == node and neighbor.f >= node.f):

return False

return True

def main():

graph = Graph()

graph.connect('Frankfurt', 'Wurzburg', 111)

graph.connect('Frankfurt', 'Mannheim', 85)

graph.connect('Wurzburg', 'Nurnberg', 104)

graph.connect('Wurzburg', 'Stuttgart', 140)

graph.connect('Wurzburg', 'Ulm', 183)

graph.connect('Mannheim', 'Nurnberg', 230)

graph.connect('Mannheim', 'Karlsruhe', 67)

graph.connect('Karlsruhe', 'Basel', 191)

graph.connect('Karlsruhe', 'Stuttgart', 64)

graph.connect('Nurnberg', 'Ulm', 171)

graph.connect('Nurnberg', 'Munchen', 170)

graph.connect('Nurnberg', 'Passau', 220)

graph.connect('Stuttgart', 'Ulm', 107)

graph.connect('Basel', 'Bern', 91)

graph.connect('Basel', 'Zurich', 85)

graph.connect('Bern', 'Zurich', 120)

graph.connect('Zurich', 'Memmingen', 184)

graph.connect('Memmingen', 'Ulm', 55)

graph.connect('Memmingen', 'Munchen', 115)

graph.connect('Munchen', 'Ulm', 123)

graph.connect('Munchen', 'Passau', 189)

graph.connect('Munchen', 'Rosenheim', 59)

graph.connect('Rosenheim', 'Salzburg', 81)

graph.connect('Passau', 'Linz', 102)

graph.connect('Salzburg', 'Linz', 126)

graph.make\_undirected()

heuristics = {}

heuristics['Basel'] = 204

heuristics['Bern'] = 247

heuristics['Frankfurt'] = 215

heuristics['Karlsruhe'] = 137

heuristics['Linz'] = 318

heuristics['Mannheim'] = 164

heuristics['Munchen'] = 120

heuristics['Memmingen'] = 47

heuristics['Nurnberg'] = 132

heuristics['Passau'] = 257

heuristics['Rosenheim'] = 168

heuristics['Stuttgart'] = 75

heuristics['Salzburg'] = 236

heuristics['Wurzburg'] = 153

heuristics['Zurich'] = 157

heuristics['Ulm'] = 0

path = best\_first\_search(graph, heuristics, 'Frankfurt', 'Ulm')

print(path)

print()

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

main()

**‘OR’**

from queue import PriorityQueue

v = 5

graph = [[] for i in range(v)]

def best\_first\_search(source, target, n):

visited = [0] \* n

visited[0] = True

pq = PriorityQueue()

pq.put((0, source))

while pq.empty() == False:

u = pq.get()[1]

print(u, end=" ")

if u == target:

break

for v, c in graph[u]:

if visited[v] == False:

visited[v] = True

pq.put((c, v))

print()

def addedge(x, y, cost):

graph[x].append((y, cost))

graph[y].append((x, cost))

addedge(0, 1, 5)

addedge(0, 2, 1)

addedge(2, 3, 2)

addedge(1, 4, 1)

addedge(3, 4, 2)

source = 0

target = 4

best\_first\_search(source, target, v)

**A\* SEARCH ALGORITHM:**

# Place the starting node in the OPEN list.

# Check if the OPEN list is empty or not, if the list is empty then return failure and stops.

# Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.

# Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.

# Return to Step 2.

from collections import deque

class Graph:

def \_\_init\_\_(self, adjacency\_list):

self.adjacency\_list = adjacency\_list

def get\_neighbors(self, v):

return self.adjacency\_list[v]

def h(self, n):

H = {

'A': 1,

'B': 1,

'C': 1,

'D': 1

}

return H[n]

def a\_star\_algorithm(self, start\_node, stop\_node):

open\_list = set([start\_node])

closed\_list = set([])

g = {}

g[start\_node] = 0

parents = {}

parents[start\_node] = start\_node

while len(open\_list) > 0:

n = None

for v in open\_list:

if n == None or g[v] + self.h(v) < g[n] + self.h(n):

n = v;

if n == None:

print('Path does not exist!')

return None

if n == stop\_node:

reconst\_path = []

while parents[n] != n:

reconst\_path.append(n)

n = parents[n]

reconst\_path.append(start\_node)

reconst\_path.reverse()

print('Path found: {}'.format(reconst\_path))

return reconst\_path

for (m, weight) in self.get\_neighbors(n):

if m not in open\_list and m not in closed\_list:

open\_list.add(m)

parents[m] = n

g[m] = g[n] + weight

else:

if g[m] > g[n] + weight:

g[m] = g[n] + weight

parents[m] = n

if m in closed\_list:

closed\_list.remove(m)

open\_list.add(m)

open\_list.remove(n)

closed\_list.add(n)

print('Path does not exist!')

return None

adjacency\_list = {

'A': [('B', 1), ('C', 3), ('D', 7)],

'B': [('D', 5)],

'C': [('D', 12)]

}

graph1 = Graph(adjacency\_list)

graph1.a\_star\_algorithm('A', 'D')

**EXPERIMENT 6 – MINIMAX ALGORITHM**

# function minimax(node, depth, Player)

# 1. if depth ==0 or node is a terminal node then return value(node)

# 2. If Player =‘Max

# set α = -∞

# for each child of node do

# value= minimax(child, depth-1, ’MIN’)

# α= max(α, Value)

# else

# set α = +∞

# for each child of node do

# value= minimax(child, depth-1, ’MAX’)

# α = min(α, Value)

import math

def minimax (curDepth, nodeIndex,

maxTurn, scores,

targetDepth):

# base case : targetDepth reached

if (curDepth == targetDepth):

return scores[nodeIndex]

if (maxTurn):

return max(minimax(curDepth + 1, nodeIndex \* 2,

False, scores, targetDepth),

minimax(curDepth + 1, nodeIndex \* 2 + 1,

False, scores, targetDepth))

else:

return min(minimax(curDepth + 1, nodeIndex \* 2,

True, scores, targetDepth),

minimax(curDepth + 1, nodeIndex \* 2 + 1,

True, scores, targetDepth))

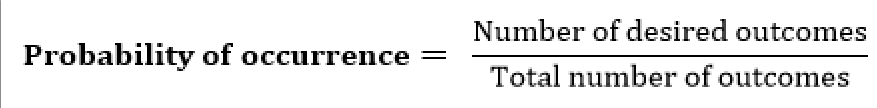
scores = [3, 5, 2, 9, 12, 5, 23, 23]

treeDepth = math.log(len(scores), 2)

print("The optimal value is : ", end = "")

print(minimax(0, 0, True, scores, treeDepth))

**EXPERIMENT 9 – UNCERATIN METHOD**

**Algorithm –**

**Code –**

**Problem 1:**

Calculate the Probability of finding how many students got the 60 marks for given data set.

import numpy as np

import collections

npArray = np.array([60, 70, 70, 70, 80, 90, 60])

c = collections.Counter(npArray)

{"value": "nbOfOccurrences"}

arraySize = npArray.size

nbOfOccurrences = c[60]

prob = (nbOfOccurrences/arraySize)\*100

print(prob)

**OUTPUT - 28.57142857142857**

**Problem 2:**

If In class 80 students and 60 students got 60 % marks then Calculate the Probability of finding how many students got the 60 marks for given data set.

import sys

Marksprob = {}

for line in sys.stdin:

line = line.strip()

ClassA, Marks = line.split('\t', 1)

def event\_probability(event\_outcomes, sample\_space):

probability = (event\_outcomes / sample\_space) \* 100

return round(probability, 1)

ClassA = 30

Marks = 15

grade\_probability = event\_probability(Marks, ClassA)

print(str(grade\_probability) + '%')

**OUTPUT – 48%**

**EXPERIMENT 10 – BLOCK WORLD (NAHI AYEGA WAISE)**

class Strips(object):

def init (self, name, preconds, effects, cost=1):

self.name = name

self.preconds = preconds

self.effects = effects

self.cost = cost

def repr (self): return self.name

class STRIPS\_domain(object):

def init (self, feats\_vals, actions):

self.feats\_vals = feats\_vals

self.actions = actions

class Planning\_problem(object):

def init (self, prob\_domain, initial\_state, goal):

self.prob\_domain = prob\_domain

self.initial\_state = initial\_state

self.goal = goal

boolean = {True, False}

def move(x,y,z):

return 'move\_'+x+'\_from\_'+y+'\_to\_'+z

def on(x):

return x+'\_is\_on'

def clear(x):

return 'clear\_'+x

def create\_blocks\_world(blocks = {'a','b','c','d'}):

blocks\_and\_table = blocks | {'table'}

stmap = {Strips(move(x,y,z),{on(x):y, clear(x):True, clear(z):True},{on(x):z, clear(y):True, clear(z):False})}

for x in blocks:

for y in blocks\_and\_table:

for z in blocks:

if x!=y and y!=z and z!=x:

stmap.update({Strips(move(x,y,'table'), {on(x):y, clear(x):True},{on(x):'table', clear(y):True})})

for x in blocks:

for y in blocks:

for z in blocks:

if x!=y:

feats\_vals = {on(x):blocks\_and\_table-{x} for x in blocks}

feats\_vals.update({clear(x):boolean for x in blocks\_and\_table})

return STRIPS\_domain(feats\_vals, stmap)

blocks1dom = create\_blocks\_world({'a','b','c'})

blocks1 = Planning\_problem(blocks1dom,

{on('a'):'table', clear('a'):True, on('b'):'c', clear('b'):True, on('c'):'table', clear('c'):False}, {on('a'):'b', on('c'):'a'})

blocks2dom = create\_blocks\_world({'a','b','c','d'})

tower4 = {clear('a'):True, on('a'):'b',

clear('b'):False, on('b'):'c',

clear('c'):False, on('c'):'d',

clear('d'):False, on('d'):'table'}

blocks2 = Planning\_problem(blocks2dom, tower4, # initial state

{on('d'):'c',on('c'):'b',on('b'):'a'})

{on('d'):'a', on('a'):'b', on('b'):'c'}

**EXPERIMENT 11 – MACHINE LEARNING ALGORITHM (JUPYTER)**

**1. LINEAR REGRESSION:**

X = [0, 6, 11, 14, 22]

Y = [1, 7, 12, 15, 21]

def best\_fit(X, Y):

    xbar = sum(X)/len(X)

    ybar = sum(Y)/len(Y)

    n = len(X)

    numer = sum([xi\*yi for xi,yi in zip(X, Y)]) - n \* xbar \* ybar

    denum = sum([xi\*\*2 for xi in X]) - n \* xbar\*\*2

    b = numer / denum

    a = ybar - b \* xbar

    print('Best Fit Line : \ny = {:.2f} + {:.2f}x'.format(a, b))

    return a,b

a, b = best\_fit(X, Y)

import matplotlib.pyplot as plt

plt.scatter(X, Y)

yfit = [a + b \* xi for xi in X]

plt.plot(X, yfit)

plt.show()

**2. KNN:**

  from scipy.spatial import distance

  class KNN():

      def fit(self, X\_train, Y\_train):

          self.X\_train = X\_train

          self.Y\_train = Y\_train

      def predict(self, X\_test):

          predictions = []

          for row in X\_test:

              label = self.closest(row)

              predictions.append(label)

          return predictions

      def closest(self, row):

          best\_dist = distance.euclidean(row, self.X\_train[0])

          best\_index = 0

          for i in range(1, len(self.X\_train)):

              dist = distance.euclidean(row, self.X\_train[i])

              if dist < best\_dist:

                  best\_dist = dist

                  best\_index = i

          return self.Y\_train[best\_index]

  from sklearn import datasets

  iris = datasets.load\_iris()

  X = iris.data

  Y = iris.target

  from sklearn.model\_selection import train\_test\_split

  X\_train, X\_test, Y\_train, Y\_test = train\_test\_split(X, Y, test\_size = .75)

  from sklearn.neighbors import KNeighborsClassifier

  classifier = KNN()

  classifier.fit(X\_train, Y\_train)

  predictions = classifier.predict(X\_test)

  from sklearn.metrics import accuracy\_score

  print (accuracy\_score(Y\_test, predictions))

**EXPERIMENT 12 - ENSEMBLE MODEL (SHAYAD NAHI AYEGA)**

from sklearn.neighbors import KNeighborsClassifier

model.fit(X, y)

predicted = model.predict(x\_test)

from sklearn.neighbors import KNeighborsClassifierdf

pd.read\_csv('iris\_df.csv')

df.columns = ['X1', 'X2', 'X3', 'X4', 'Y']

df = df.drop(['X4', 'X3'], 1)

df.head()

sns.set\_context('notebook', font\_scale=1.1)

sns.set\_style('ticks')

sns.lmplot('X1','X2', scatter=True, fit\_reg=False, data=df, hue='Y')

plt.ylabel('X2')

plt.xlabel('X1')

from sklearn.cross\_validation import train\_test\_split

neighbors = KNeighborsClassifier(n\_neighbors=5)

X = df.values[:, 0:2]

Y = df.values[:, 2]

trainX, testX, trainY, testY = train\_test\_split( X, Y, test\_size = 0.3)

neighbors.fit(trainX, trainY)

print('Accuracy: \n', neighbors.score(testX, testY))

pred = neighbors.predict(testX)

**TOWER OF HANOI:**

# START

# Procedure Hanoi(disk, source, dest, aux)

# IF disk == 1, THEN

# move disk from source to dest

# ELSE

# Hanoi(disk - 1, source, aux, dest) // Step 1

# move disk from source to dest // Step 2

# Hanoi(disk - 1, aux, dest, source) // Step 3

# END IF

# END Procedure

# STOP

def tower\_of\_hanoi(disks, source, auxiliary, target):

if(disks == 1):

print('MOVE DISK 1 FROM ROD {} TO ROD {}.'.format(source, target))

return

tower\_of\_hanoi(disks - 1, source, target, auxiliary)

print('MOVE DISK {} FROM ROD {} TO ROD {}.'.format(disks, source, target))

tower\_of\_hanoi(disks - 1, auxiliary, source, target)

disks = int(input('ENTER THE NUMBER OF DISKS = '))

tower\_of\_hanoi(disks, 'A', 'B', 'C')

**TIC TAC TOE:**

theBoard = {'7': ' ' , '8': ' ' , '9': ' ' ,

'4': ' ' , '5': ' ' , '6': ' ' ,

'1': ' ' , '2': ' ' , '3': ' ' }

board\_keys = []

for key in theBoard:

board\_keys.append(key)

def printBoard(board):

print(board['7'] + '|' + board['8'] + '|' + board['9'])

print('-+-+-')

print(board['4'] + '|' + board['5'] + '|' + board['6'])

print('-+-+-')

print(board['1'] + '|' + board['2'] + '|' + board['3'])

def game():

turn = 'X'

count = 0

for i in range(10):

printBoard(theBoard)

print("It's your turn," + turn + ".Move to which place?")

move = input()

if theBoard[move] == ' ':

theBoard[move] = turn

count += 1

else:

print("That place is already filled.\nMove to which place?")

continue

if count >= 5:

if theBoard['7'] == theBoard['8'] == theBoard['9'] != ' ': # across the top

printBoard(theBoard)

print("\nGame Over.\n")

print(" \*\*\*\* " +turn + " won. \*\*\*\*")

break

elif theBoard['4'] == theBoard['5'] == theBoard['6'] != ' ': # across the middle

printBoard(theBoard)

print("\nGame Over.\n")

print(" \*\*\*\* " +turn + " won. \*\*\*\*")

break

elif theBoard['1'] == theBoard['2'] == theBoard['3'] != ' ': # across the bottom

printBoard(theBoard)

print("\nGame Over.\n")

print(" \*\*\*\* " +turn + " won. \*\*\*\*")

break

elif theBoard['1'] == theBoard['4'] == theBoard['7'] != ' ': # down the left side

printBoard(theBoard)

print("\nGame Over.\n")

print(" \*\*\*\* " +turn + " won. \*\*\*\*")

break

elif theBoard['2'] == theBoard['5'] == theBoard['8'] != ' ': # down the middle

printBoard(theBoard)

print("\nGame Over.\n")

print(" \*\*\*\* " +turn + " won. \*\*\*\*")

break

elif theBoard['3'] == theBoard['6'] == theBoard['9'] != ' ': # down the right side

printBoard(theBoard)

print("\nGame Over.\n")

print(" \*\*\*\* " +turn + " won. \*\*\*\*")

break

elif theBoard['7'] == theBoard['5'] == theBoard['3'] != ' ': # diagonal

printBoard(theBoard)

print("\nGame Over.\n")

print(" \*\*\*\* " +turn + " won. \*\*\*\*")

break

elif theBoard['1'] == theBoard['5'] == theBoard['9'] != ' ': # diagonal

printBoard(theBoard)

print("\nGame Over.\n")

print(" \*\*\*\* " +turn + " won. \*\*\*\*")

break

if count == 9:

print("\nGame Over.\n")

print("It's a Tie!!")

if turn =='X':

turn = 'O'

else:

turn = 'X'

restart = input("Do want to play Again?(y/n)")

if restart == "y" or restart == "Y":

for key in board\_keys:

theBoard[key] = " "

game()

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

game()

**N-PUZZLE:**

def print\_in\_format(matrix):

for i in range(9):

if i % 3 == 0 and i > 0:

print("")

print(str(matrix[i])+" ", end="")

def count(s):

c = 0

ideal = [1, 2, 3,

4, 5, 6,

7, 8, 0]

for i in range(9):

if s[i] != 0 and s[i] != ideal[i]:

c += 1

return c

def move(ar, p, st):

rh = 999999

store\_st = st.copy()

for i in range(len(ar)):

dupl\_st = st.copy()

temp = dupl\_st[p]

dupl\_st[p] = dupl\_st[arr[i]]

dupl\_st[arr[i]] = temp

tmp\_rh = count(dupl\_st)

if tmp\_rh < rh:

rh = tmp\_rh

store\_st = dupl\_st.copy()

return store\_st, rh

state = [1, 2, 3,

0, 5, 6,

4, 7, 8]

h = count(state)

Level = 1

print("\n------ Level "+str(Level)+" ------")

print\_in\_format(state)

print("\nHeuristic Value(Misplaced) : "+str(h))

while h > 0:

pos = int(state.index(0))

Level += 1

if pos == 0:

arr = [1, 3]

state, h = move(arr, pos, state)

elif pos == 1:

arr = [0, 2, 4]

state, h = move(arr, pos, state)

elif pos == 2:

arr = [1, 5]

state, h = move(arr, pos, state)

elif pos == 3:

arr = [0, 4, 6]

state, h = move(arr, pos, state)

elif pos == 4:

arr = [1, 3, 5, 7]

state, h = move(arr, pos, state)

elif pos == 5:

arr = [2, 4, 8]

state, h = move(arr, pos, state)

elif pos == 6:

arr = [3, 7]

state, h = move(arr, pos, state)

elif pos == 7:

arr = [4, 6, 8]

state, h = move(arr, pos, state)

elif pos == 8:

arr = [5, 6]

state, h = move(arr, pos, state)

print("\n------ Level "+str(Level)+" ------")

print\_in\_format(state)

print("\nHeuristic Value(Misplaced) : "+str(h))

**RIVER CROSSING PUZZLE:**

def print\_in\_format(matrix):

for i in range(9):

if i % 3 == 0 and i > 0:

print("")

print(str(matrix[i])+" ", end="")

def count(s):

c = 0

ideal = [1, 2, 3,

4, 5, 6,

7, 8, 0]

for i in range(9):

if s[i] != 0 and s[i] != ideal[i]:

c += 1

return c

def move(ar, p, st):

rh = 999999

store\_st = st.copy()

for i in range(len(ar)):

dupl\_st = st.copy()

temp = dupl\_st[p]

dupl\_st[p] = dupl\_st[arr[i]]

dupl\_st[arr[i]] = temp

tmp\_rh = count(dupl\_st)

if tmp\_rh < rh:

rh = tmp\_rh

store\_st = dupl\_st.copy()

return store\_st, rh

state = [1, 2, 3,

0, 5, 6,

4, 7, 8]

h = count(state)

Level = 1

print("\n------ Level "+str(Level)+" ------")

print\_in\_format(state)

print("\nHeuristic Value(Misplaced) : "+str(h))

while h > 0:

pos = int(state.index(0))

Level += 1

if pos == 0:

arr = [1, 3]

state, h = move(arr, pos, state)

elif pos == 1:

arr = [0, 2, 4]

state, h = move(arr, pos, state)

elif pos == 2:

arr = [1, 5]

state, h = move(arr, pos, state)

elif pos == 3:

arr = [0, 4, 6]

state, h = move(arr, pos, state)

elif pos == 4:

arr = [1, 3, 5, 7]

state, h = move(arr, pos, state)

elif pos == 5:

arr = [2, 4, 8]

state, h = move(arr, pos, state)

elif pos == 6:

arr = [3, 7]

state, h = move(arr, pos, state)

elif pos == 7:

arr = [4, 6, 8]

state, h = move(arr, pos, state)

elif pos == 8:

arr = [5, 6]

state, h = move(arr, pos, state)

print("\n------ Level "+str(Level)+" ------")

print\_in\_format(state)

print("\nHeuristic Value(Misplaced) : "+str(h))

import os

import time

names = {"F": "Farmer",

"W": "Wolf",

"G": "Goat",

"C": "Cabbage"}

forbidden\_states = [{"W", "G"}, {"G", "C"}, {"G", "C", "W"}]

def clear():

print("\*" \* 60, "\n")

def print\_state(state):

left\_bank, right\_bank = state

print("#### CURRENT STATE OF PUZZLE ####")

print()

left\_bank\_display = [names[item] for item in left\_bank]

right\_bank\_display = [names[item] for item in right\_bank]

print(left\_bank\_display, "|", right\_bank\_display if right\_bank else "[]")

print()

def get\_move():

print("Which item do you wish to take across the river?")

answer = ""

while answer.upper() not in ["F", "W", "G", "C"]:

answer = input("Just Farmer (f), Wolf (w), Goat (g) or Cabbage (c)? ")

return answer.upper()

def process\_move(move, state):

temp\_state = [state[0].copy(), state[1].copy()]

containing\_set = 0 if move in state[0] else 1

if "F" not in state[containing\_set]:

print("Illegal move.")

print()

time.sleep(1)

return state

if containing\_set == 0:

temp\_state[0].difference\_update({move, "F"})

temp\_state[1].update([move, "F"])

elif containing\_set == 1:

temp\_state[1].difference\_update({move, "F"})

temp\_state[0].update([move, "F"])

if temp\_state[0] not in forbidden\_states and temp\_state[1] not in forbidden\_states:

state = [temp\_state[0].copy(), temp\_state[1].copy()]

else:

print("Illegal move.")

print()

time.sleep(1)

print()

return state

def is\_win(state):

return state[1] == {"F", "W", "G", "C"}

def main():

left\_bank = {"F", "W", "G", "C"}

right\_bank = set()

state = [left\_bank, right\_bank]

while not is\_win(state):

clear()

print\_state(state)

move = get\_move()

state = process\_move(move, state)

print("Well done - you solved the puzzle!")

main()