**1.AIM: Write a program to implement BFS Traversal.**

**Source Code:**

from collections import defaultdict

class Graph:

      def \_\_init\_\_(self):

            self. graph = defaultdict(list)

      def add Edge(self,u,v):

        self. graph[u].append(v)

    def BFS(self, s):

          visited = [False] \* (len(self. graph))

        queue = []

        # Mark the source node as

        # visited and enqueue it

        queue.append(s)

        visited[s] = True

        while queue:

            # Dequeue a vertex from

            # queue and print it

            s = queue.pop(0)

            print (s, end = " ")

            # Get all adjacent vertices of the

            # dequeued vertex s. If a adjacent

            # has not been visited, then mark it

            # visited and enqueue it

            for i in self.graph[s]:

                if visited[i] == False:

                    queue.append(i)

                    visited[i] = True

# Driver code

# Create a graph given in

# the above diagram

g = Graph()

g.addEdge (0, 1)

g.addEdge (0, 2)

g.addEdge (1, 2)

g.addEdge (2, 0)

g.addEdge (2, 3)

g.addEdge (3, 3)

print ("Following is Breadth First Traversal"

                  " (starting from vertex 2)")

g.BFS(2)

**OUTPUT:**

Following is Breadth First Traversal (starting from vertex 2)

2 0 3 1

**2. AIM: Write a program to implement DFS Traversal.**

**SOURCE CODE:**

# Python3 program to print DFS traversal

# from a given given graph

from collections import defaultdict

# This class represents a directed graph using

# adjacency list representation

class Graph:

    # Constructor

    def \_\_init\_\_(self):

        # default dictionary to store graph

        self. graph = defaultdict(list)

    # function to add an edge to graph

    def add Edge(self, u, v):

        self.graph[u].append(v)

    # A function used by DFS

    def DFSUtil(self, v, visited):

        # Mark the current node as visited

        # and print it

        visited.add(v)

        print(v, end=' ')

        # Recur for all the vertices

        # adjacent to this vertex

        for neighbour in self.graph[v]:

            if neighbour not in visited:

                self.DFSUtil(neighbour, visited)

    # The function to do DFS traversal. It uses

    # recursive DFSUtil()

    def DFS(self, v):

        # Create a set to store visited vertices

        visited = set()

        # Call the recursive helper function

        # to print DFS traversal

        self.DFSUtil(v, visited)

# Driver code

# Create a graph given

# in the above diagram

g = Graph()

g.addEdge(0, 1)

g.addEdge(0, 2)

g.addEdge(1, 2)

g.addEdge(2, 0)

g.addEdge(2, 3)

g.addEdge(3, 3)

print("Following is DFS from (starting from vertex 2)")

g.DFS(2)

**OUTPUT:**

Following is DFS from (starting from vertex 2)

1. 0 1 3

**3.AIM:Write a program to implement A\* Search.**

**SOURCE CODE:**

# This class represent a graph

class Graph:

# Initialize the class

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

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

self.directed = directed

if not directed:

self.make\_undirected()

# Create an undirected graph by adding symmetric edges

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

# Add a link from A and B of given distance, and also add the inverse link if the graph is undirected

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

# Get neighbors or a neighbor

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

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

if b is None:

return links

else:

return links.get(b)

# Return a list of nodes in the graph

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)

# This class represent a node

class Node:

# Initialize the class

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

self.name = name

self.parent = parent

self.g = 0 # Distance to start node

self.h = 0 # Distance to goal node

self.f = 0 # Total cost

# Compare nodes

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

return self.name == other.name

# Sort nodes

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

return self.f < other.f

# Print node

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

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

# A\* search

def astar\_search(graph, heuristics, start, end):

# Create lists for open nodes and closed nodes

open = []

closed = []

# Create a start node and an goal node

start\_node = Node(start, None)

goal\_node = Node(end, None)

# Add the start node

open.append(start\_node)

# Loop until the open list is empty

while len(open) > 0:

# Sort the open list to get the node with the lowest cost first

open.sort()

# Get the node with the lowest cost

current\_node = open.pop(0)

# Add the current node to the closed list

closed.append(current\_node)

# Check if we have reached the goal, return the path

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 reversed path

return path[::-1]

# Get neighbours

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

# Loop neighbors

for key, value in neighbors.items():

# Create a neighbor node

neighbor = Node(key, current\_node)

# Check if the neighbor is in the closed list

if(neighbor in closed):

continue

# Calculate full path cost

neighbor.g = current\_node.g + graph.get(current\_node.name, neighbor.name)

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

neighbor.f = neighbor.g + neighbor.h

# Check if neighbor is in open list and if it has a lower f value

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

# Everything is green, add neighbor to open list

open.append(neighbor)

# Return None, no path is found

return None

# Check if a neighbor should be added to open list

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

for node in open:

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

return False

return True

# The main entry point for this module

def main():

# Create a graph

graph = Graph()

# Create graph connections (Actual distance)

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)

# Make graph undirected, create symmetric connections

graph.make\_undirected()

# Create heuristics (straight-line distance, air-travel distance)

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

# Run the search algorithm

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

print(path)

print()

# Tell python to run main method

if \_\_name\_\_ == "\_\_main\_\_": main()

**OUTPUT:**

['Frankfurt: 0', 'Wurzburg: 111', 'Ulm: 294']

**4.AIM:Write a program to implement Travelling Salesman Problem.**

**SOURCE CODE:**

# Python3 program to implement traveling salesman

# problem using naive approach.

from sys import maxsize

from itertools import permutations

V = 4

# implementation of traveling Salesman Problem

def travellingSalesmanProblem(graph, s):

    # store all vertex apart from source vertex

    vertex = []

    for i in range(V):

        if i != s:

            vertex.append(i)

    # store minimum weight Hamiltonian Cycle

    min\_path = maxsize

    next\_permutation=permutations(vertex)

    for i in next\_permutation:

        # store current Path weight(cost)

        current\_pathweight = 0

        # compute current path weight

        k = s

        for j in i:

            current\_pathweight += graph[k][j]

            k = j

        current\_pathweight += graph[k][s]

        # update minimum

        min\_path = min(min\_path, current\_pathweight)

    return min\_path

# Driver Code

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

    # matrix representation of graph

    graph = [[0, 10, 15, 20], [10, 0, 35, 25],

            [15, 35, 0, 30], [20, 25, 30, 0]]

    s = 0

    print(travellingSalesmanProblem(graph, s))

**OUTPUT:**

**80**

**5.AIM:Write a program to implement Graph Coloring Problem.**

**SOURCE CODE:**

#!/usr/bin/env python

# coding: utf-8

# In[6]:

colors = ['Red', 'Blue', 'Green', 'Yellow', 'Black']

states = ['Andhra', 'Karnataka', 'TamilNadu', 'Kerala']

neighbors = {}

neighbors['Andhra'] = ['Karnataka', 'TamilNadu']

neighbors['Karnataka'] = ['Andhra', 'TamilNadu', 'Kerala']

neighbors['TamilNadu'] = ['Andhra', 'Karnataka', 'Kerala']

neighbors['Kerala'] = ['Karnataka', 'TamilNadu']

colors\_of\_states = {}

def promising(state, color):

    for neighbor in neighbors.get(state):

        color\_of\_neighbor = colors\_of\_states.get(neighbor)

        if color\_of\_neighbor == color:

            return False

    return True

def get\_color\_for\_state(state):

    for color in colors:

        if promising(state, color):

            return color

def main():

    for state in states:

        colors\_of\_states[state] = get\_color\_for\_state(state)

    print (colors\_of\_states)

main()

# In[ ]:

# In[ ]:

**OUTPUT:** {'Andhra': 'Red', 'Karnataka': 'Blue', 'TamilNadu': 'Green', 'Kerala': 'Red'}

**6.AIM:Write a program to implement Missionaries and Cannibals Problem.**

**SOURCE CODE:**

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()

**OUTPUT:**

Missionaries and Cannibals solution:

(cannibalLeft,missionaryLeft,boat,cannibalRight,missionaryRight)

(3,3,left,0,0)

(1,3,right,2,0)

(2,3,left,1,0)

(0,3,right,3,0)

(1,3,left,2,0)

(1,1,right,2,2)

(2,2,left,1,1)

(2,0,right,1,3)

(3,0,left,0,3)

(1,0,right,2,3)

(1,1,left,2,2)

(0,0,right,3,3)

**7. AIM: Write a program to implement Water Jug Problem.**

**SOURCE CODE:**

#here we are using default dict which is used to initialize the dict elements with default value

from collections import defaultdict

#jug1 and jug2 have the max capacity of two jugs and target is the amount to be measured

jug1=int(input("Enter the capacity of Jug1"))

jug2=int(input("Enter the capacity of Jug2"))

target=int(input("Enter the capacity to be measured within the range of Max(Jug1,Jug2)"))

# Initialize dictionary with

# default value as false.

visited = defaultdict(lambda: False)

print(visited)

# Recursive function which prints the intermediate steps to reach the final

# solution and return boolean value  (True if solution is possible, otherwise False).

# amountt1 and amount2 are the amount of water present  in both jugs at a certain point of time.

def waterjugProblem(amount1, amount2):

    # returns true if out goal is achieved else return False

    if (amount1 == target and amount2 == 0) or (amount2 == target and amount1 == 0):

        print(amount1,amount2)

        return True

    # Checks if we have already visited the

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

    if visited[(amount1, amount2)] == False:

        print(amount1,amount2)

        # Changes the boolean value to True indicating we already visited this combination

        visited[(amount1, amount2)] = True

        # here we are exploring 6 possible Solution

        #Empty the first jug completely

        #Empty the second jug completely

        #Fill the first jug

        #Fill the second jug

        #Fill the water from the second jug into the first jug until the first jug is full or the second jug has no water left

        #Fill the water from the first jug into the second jug until the second jug is full or the first jug has no water left1.

        return (waterjugProblem(0, amount2) or

                waterjugProblem(amount1, 0) or

                waterjugProblem(jug1, amount2) or

                waterjugProblem(amount1, jug2) or

                waterjugProblem(amount1 + min(amount2, (jug1-amount1)),

                amount2 - min(amount2, (jug1-amount1))) or

                waterjugProblem(amount1 - min(amount1, (jug2-amount2)),

                amount2 + min(amount1, (jug2-amount2))))

    #if the combination (amount1,amount2) is already visited return False as we do not perform same resursive call

    else:

        return False

print("Steps: ")

# Call the function and pass the

# initial amount of water present in both jugs.

waterjugProblem(0, 0)

**OUTPUT:**

JUG1 (5) JUG2 (7)

0 0

5 0

0 5

5 5

3 7

0 3

5 3

1 7

0 1

5 1

0 6

5 6

4 7

0 4

**8. AIM:Write a program to implement Hangman game.**

**SOURCE CODE:**

import random

word\_list = ["insert", "your", "words", "in", "this", "python", "list"]

def get\_word(word\_list):

word = random.choice(word\_list)

return word.upper()

def play(word):

word\_completion = "\_" \* len(word)

guessed = False

guessed\_letters = []

guessed\_words = []

tries = 6

print("Let's play Hangman")

print(display\_hangman(tries))

print(word\_completion)

print("\n")

while not guessed and tries > 0:

guess = input("guess a letter or word: ").upper()

if len(guess) == 1 and guess.isalpha():

if guess in guessed\_letters:

print("you already tried", guess, "!")

elif guess not in word:

print(guess, "isn't in the word :(")

tries -= 1

guessed\_letters.append(guess)

else:

print("Nice one,", guess, "is in the word!")

guessed\_letters.append(guess)

word\_as\_list = list(word\_completion)

indices = [i for i, letter in enumerate(word) if letter == guess]

for index in indices:

word\_as\_list[index] = guess

word\_completion = "".join(word\_as\_list)

if "\_" not in word\_completion:

guessed = True

elif len(guess) == len(word) and guess.isalpha():

if guess in guessed\_words:

print("You already tried ", guess, "!")

elif guess != word:

print(guess, " ist nicht das Wort :(")

tries -= 1

guessed\_words.append(guess)

else:

guessed = True

word\_completion = word

else:

print("invalid input")

print(display\_hangman(tries))

print(word\_completion)

print("\n")

if guessed:

print("Good Job, you guessed the word!")

else:

print("I'm sorry, but you ran out of tries. The word was " + word + ". Maybe next time!"

def display\_hangman(tries):

stages = [ """

--------

| |

| O

| \\|/

| |

| / \\

-

""",

"""

--------

| |

| O

| \\|/

| |

| /

-

""",

"""

--------

| |

| O

| \\|/

| |

|

-

""",

"""

--------

| |

| O

| \\|

| |

|

-

""",

"""

--------

| |

| O

| |

| |

|

-

""",

"""

--------

| |

| O

|

|

|

-

""",

"""

--------

| |

|

|

|

|

-

"""

]

return stages[tries]

def main():

word = get\_word(word\_list)

play(word)

while input("Again? (Y/N) ").upper() == "Y":

word = get\_word(word\_list)

play(word)

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

main()

**OUTPUT:**

Let's play Hangman

--------

| |

|

|

|

|

-

\_\_\_\_\_\_

guess a letter or word: t

Nice one, T is in the word!

--------

| |

|

|

|

|

-

\_\_\_\_\_T

guess a letter or word: h

H isn't in the word :(

--------

| |

| O

|

|

|

-

\_\_\_\_\_T

guess a letter or word: i

Nice one, I is in the word!

--------

| |

| O

|

|

|

-

I\_\_\_\_T

guess a letter or word: s

Nice one, S is in the word!

--------

| |

| O

|

|

|

-

I\_S\_\_T

guess a letter or word:

**9. AIM:Write a program to implement Tic-Tac-Toe game.**

**SOURCE CODE:**

# Tic Tac Toe

import random

def drawBoard(board):

# This function prints out the board that it was passed.

# "board" is a list of 10 strings representing the board (ignore index 0)

print(' | |')

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

print(' | |')

print('-----------')

print(' | |')

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

print(' | |')

print('-----------')

print(' | |')

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

print(' | |')

def inputPlayerLetter():

# Lets the player type which letter they want to be.

# Returns a list with the player's letter as the first item, and the computer's letter as the second.

letter = ''

while not (letter == 'X' or letter == 'O'):

print('Do you want to be X or O?')

letter = input().upper()

# the first element in the tuple is the player's letter, the second is the computer's letter.

if letter == 'X':

return ['X', 'O']

else:

return ['O', 'X']

def whoGoesFirst():

# Randomly choose the player who goes first.

if random.randint(0, 1) == 0:

return 'computer'

else:

return 'player'

def playAgain():

# This function returns True if the player wants to play again, otherwise it returns False.

print('Do you want to play again? (yes or no)')

return input().lower().startswith('y')

def makeMove(board, letter, move):

board[move] = letter

def isWinner(bo, le):

# Given a board and a player's letter, this function returns True if that player has won.

# We use bo instead of board and le instead of letter so we don't have to type as much.

return ((bo[7] == le and bo[8] == le and bo[9] == le) or # across the top

(bo[4] == le and bo[5] == le and bo[6] == le) or # across the middle

(bo[1] == le and bo[2] == le and bo[3] == le) or # across the bottom

(bo[7] == le and bo[4] == le and bo[1] == le) or # down the left side

(bo[8] == le and bo[5] == le and bo[2] == le) or # down the middle

(bo[9] == le and bo[6] == le and bo[3] == le) or # down the right side

(bo[7] == le and bo[5] == le and bo[3] == le) or # diagonal

(bo[9] == le and bo[5] == le and bo[1] == le)) # diagonal

def getBoardCopy(board):

# Make a duplicate of the board list and return it the duplicate.

dupeBoard = []

for i in board:

dupeBoard.append(i)

return dupeBoard

def isSpaceFree(board, move):

# Return true if the passed move is free on the passed board.

return board[move] == ' '

def getPlayerMove(board):

# Let the player type in his move.

move = ' '

while move not in '1 2 3 4 5 6 7 8 9'.split() or not isSpaceFree(board, int(move)):

print('What is your next move? (1-9)')

move = input()

return int(move)

def chooseRandomMoveFromList(board, movesList):

# Returns a valid move from the passed list on the passed board.

# Returns None if there is no valid move.

possibleMoves = []

for i in movesList:

if isSpaceFree(board, i):

possibleMoves.append(i)

if len(possibleMoves) != 0:

return random.choice(possibleMoves)

else:

return None

def getComputerMove(board, computerLetter):

# Given a board and the computer's letter, determine where to move and return that move.

if computerLetter == 'X':

playerLetter = 'O'

else:

playerLetter = 'X'

# Here is our algorithm for our Tic Tac Toe AI:

# First, check if we can win in the next move

for i in range(1, 10):

copy = getBoardCopy(board)

if isSpaceFree(copy, i):

makeMove(copy, computerLetter, i)

if isWinner(copy, computerLetter):

return i

# Check if the player could win on his next move, and block them.

for i in range(1, 10):

copy = getBoardCopy(board)

if isSpaceFree(copy, i):

makeMove(copy, playerLetter, i)

if isWinner(copy, playerLetter):

return i

# Try to take one of the corners, if they are free.

move = chooseRandomMoveFromList(board, [1, 3, 7, 9])

if move != None:

return move

# Try to take the center, if it is free.

if isSpaceFree(board, 5):

return 5

# Move on one of the sides.

return chooseRandomMoveFromList(board, [2, 4, 6, 8])

def isBoardFull(board):

# Return True if every space on the board has been taken. Otherwise return False.

for i in range(1, 10):

if isSpaceFree(board, i):

return False

return True

print('Welcome to Tic Tac Toe!')

while True:

# Reset the board

theBoard = [' '] \* 10

playerLetter, computerLetter = inputPlayerLetter()

turn = whoGoesFirst()

print('The ' + turn + ' will go first.')

gameIsPlaying = True

while gameIsPlaying:

if turn == 'player':

# Player's turn.

drawBoard(theBoard)

move = getPlayerMove(theBoard)

makeMove(theBoard, playerLetter, move)

if isWinner(theBoard, playerLetter):

drawBoard(theBoard)

print('Hooray! You have won the game!')

gameIsPlaying = False

else:

if isBoardFull(theBoard):

drawBoard(theBoard)

print('The game is a tie!')

break

else:

turn = 'computer'

else:

# Computer's turn.

move = getComputerMove(theBoard, computerLetter)

makeMove(theBoard, computerLetter, move)

if isWinner(theBoard, computerLetter):

drawBoard(theBoard)

print('The computer has beaten you! You lose.')

gameIsPlaying = False

else:

if isBoardFull(theBoard):

drawBoard(theBoard)

print('The game is a tie!')

break

else:

turn = 'player'

if not playAgain():

break

**OUTPUT:**

Welcome to Tic Tac Toe!

Do you want to be X or O?

X

The player will go first.

| |

| |

| |

-----------

| |

| |

| |

-----------

| |

| |

| |

What is your next move? (1-9)

5

| |

| |

| |

-----------

| |

| X |

| |

-----------

| |

| | O

| |

What is your next move? (1-9)

2

| |

| O |

| |

-----------

| |

| X |

| |

-----------

| |

| X | O

| |

What is your next move? (1-9)

4

| |

| O |

| |

-----------

| |

X | X | O

| |

-----------

| |

| X | O

| |

What is your next move? (1-9)

3

What is your next move? (1-9)

7

| |

X | O | O

| |

-----------

| |

X | X | O

| |

-----------

| |

| X | O

| |

The computer has beaten you! You lose.

Do you want to play again? (yes or no)

**10.AIM:Write a program to implement 8 Queens Problem**

**SOURCE CODE:**

#Number of queens

print ("Enter the number of queens")

N = int(input())

#chessboard

#NxN matrix with all elements 0

board = [[0]\*N for \_ in range(N)]

def is\_attack(i, j):

#checking if there is a queen in row or column

for k in range(0,N):

if board[i][k]==1 or board[k][j]==1:

return True

#checking diagonals

for k in range(0,N):

for l in range(0,N):

if (k+l==i+j) or (k-l==i-j):

if board[k][l]==1:

return True

return False

def N\_queen(n):

#if n is 0, solution found

if n==0:

return True

for i in range(0,N):

for j in range(0,N):

'''checking if we can place a queen here or not

queen will not be placed if the place is being attacked

or already occupied'''

if (not(is\_attack(i,j))) and (board[i][j]!=1):

board[i][j] = 1

#recursion

#wether we can put the next queen with this arrangment or not

if N\_queen(n-1)==True:

return True

board[i][j] = 0

return False

N\_queen(N)

for i in board:

print (i)

**OUTPUT:**

Enter the number of queens

8

[1, 0, 0, 0, 0, 0, 0, 0]

[0, 0, 0, 0, 1, 0, 0, 0]

[0, 0, 0, 0, 0, 0, 0, 1]

[0, 0, 0, 0, 0, 1, 0, 0]

[0, 0, 1, 0, 0, 0, 0, 0]

[0, 0, 0, 0, 0, 0, 1, 0]

[0, 1, 0, 0, 0, 0, 0, 0]

[0, 0, 0, 1, 0, 0, 0, 0]

Enter the number of queens

4

[0, 1, 0, 0]

[0, 0, 0, 1]

[1, 0, 0, 0]

[0, 0, 1, 0]

**11.AIM:Write a program to implement Bayesian Network.**

**SOURCE CODE:**

**Install Pomegranate Library and watermark library**

!pip install pomegranate

!pip install watermark

Requirement already satisfied: pomegranate in /usr/local/lib/python3.6/dist-packages (0.14.1)

Requirement already satisfied: numpy>=1.8.0 in /usr/local/lib/python3.6/dist-packages (from pomegranate) (1.19.5)

Requirement already satisfied: joblib>=0.9.0b4 in /usr/local/lib/python3.6/dist-packages (from pomegranate) (1.0.0)

Requirement already satisfied: pyyaml in /usr/local/lib/python3.6/dist-packages (from pomegranate) (3.13)

Requirement already satisfied: networkx>=2.0 in /usr/local/lib/python3.6/dist-packages (from pomegranate) (2.5)

Requirement already satisfied: scipy>=0.17.0 in /usr/local/lib/python3.6/dist-packages (from pomegranate) (1.4.1)

Requirement already satisfied: decorator>=4.3.0 in /usr/local/lib/python3.6/dist-packages (from networkx>=2.0->pomegranate) (4.4.2)

**Collecting watermark**

**Downloading** <https://files.pythonhosted.org/packages/60/fe/3ed83b6122e70dce6fe269dfd763103c333f168bf91037add73ea4fe81c2/watermark-2.0.2-py2.py3-none-any.whl>

Requirement already satisfied: ipython in /usr/local/lib/python3.6/dist-packages (from watermark) (5.5.0)

Requirement already satisfied: simplegeneric>0.8 in /usr/local/lib/python3.6/dist-packages (from ipython->watermark) (0.8.1)

Requirement already satisfied: pexpect; sys\_platform != "win32" in /usr/local/lib/python3.6/dist-packages (from ipython->watermark) (4.8.0)

Requirement already satisfied: pygments in /usr/local/lib/python3.6/dist-packages (from ipython->watermark) (2.6.1)

Requirement already satisfied: decorator in /usr/local/lib/python3.6/dist-packages (from ipython->watermark) (4.4.2)

Requirement already satisfied: pickleshare in /usr/local/lib/python3.6/dist-packages (from ipython->watermark) (0.7.5)

Requirement already satisfied: prompt-toolkit<2.0.0,>=1.0.4 in /usr/local/lib/python3.6/dist-packages (from ipython->watermark) (1.0.18)

Requirement already satisfied: setuptools>=18.5 in /usr/local/lib/python3.6/dist-packages (from ipython->watermark) (53.0.0)

Requirement already satisfied: traitlets>=4.2 in /usr/local/lib/python3.6/dist-packages (from ipython->watermark) (4.3.3)

Requirement already satisfied: ptyprocess>=0.5 in /usr/local/lib/python3.6/dist-packages (from pexpect; sys\_platform != "win32"->ipython->watermark) (0.7.0)

Requirement already satisfied: wcwidth in /usr/local/lib/python3.6/dist-packages (from prompt-toolkit<2.0.0,>=1.0.4->ipython->watermark) (0.2.5)

Requirement already satisfied: six>=1.9.0 in /usr/local/lib/python3.6/dist-packages (from prompt-toolkit<2.0.0,>=1.0.4->ipython->watermark) (1.15.0)

Requirement already satisfied: ipython-genutils in /usr/local/lib/python3.6/dist-packages (from traitlets>=4.2->ipython->watermark) (0.2.0)

Installing collected packages: watermark

Successfully installed watermark-2.0.2

**Initialization**

%matplotlib inline

import matplotlib.pyplot as plt

import seaborn; seaborn.set\_style('whitegrid')

import numpy

from pomegranate import \*

numpy.random.seed(0)

numpy.set\_printoptions(suppress=True)

%load\_ext watermark

%watermark -m -n -p numpy,scipy,pomegranate

Tue Feb 09 2021

numpy 1.19.5

scipy 1.4.1

pomegranate 0.14.1

compiler : GCC 8.4.0

system : Linux

release : 4.19.112+

machine : x86\_64

processor : x86\_64

CPU cores : 2

interpreter: 64bit

**The Monty Hall Gameshow**

# The guests initial door selection is completely random

guest = DiscreteDistribution({'A': 1./3, 'B': 1./3, 'C': 1./3})

# The door the prize is behind is also completely random

prize = DiscreteDistribution({'A': 1./3, 'B': 1./3, 'C': 1./3})

    # Monty is dependent on both the guest and the prize.

monty = ConditionalProbabilityTable(

        [[ 'A', 'A', 'A', 0.0 ],

         [ 'A', 'A', 'B', 0.5 ],

         [ 'A', 'A', 'C', 0.5 ],

         [ 'A', 'B', 'A', 0.0 ],

         [ 'A', 'B', 'B', 0.0 ],

         [ 'A', 'B', 'C', 1.0 ],

         [ 'A', 'C', 'A', 0.0 ],

         [ 'A', 'C', 'B', 1.0 ],

         [ 'A', 'C', 'C', 0.0 ],

         [ 'B', 'A', 'A', 0.0 ],

         [ 'B', 'A', 'B', 0.0 ],

         [ 'B', 'A', 'C', 1.0 ],

         [ 'B', 'B', 'A', 0.5 ],

         [ 'B', 'B', 'B', 0.0 ],

         [ 'B', 'B', 'C', 0.5 ],

         [ 'B', 'C', 'A', 1.0 ],

         [ 'B', 'C', 'B', 0.0 ],

         [ 'B', 'C', 'C', 0.0 ],

         [ 'C', 'A', 'A', 0.0 ],

         [ 'C', 'A', 'B', 1.0 ],

         [ 'C', 'A', 'C', 0.0 ],

         [ 'C', 'B', 'A', 1.0 ],

         [ 'C', 'B', 'B', 0.0 ],

         [ 'C', 'B', 'C', 0.0 ],

         [ 'C', 'C', 'A', 0.5 ],

         [ 'C', 'C', 'B', 0.5 ],

         [ 'C', 'C', 'C', 0.0 ]], [guest, prize])

**State objects hold both the distribution, and a high level name.**

s1 = State(guest, name="guest")

s2 = State(prize, name="prize")

s3 = State(monty, name="monty")

**Create Bayesian Network**

# Create the Bayesian network object with a useful name

model = BayesianNetwork("Monty Hall Problem")

# Add the three states to the network

model.add\_states(s1, s2, s3)

**Add edges which represent conditional dependencies, where the second node is**

model.add\_edge(s1, s3)

model.add\_edge(s2, s3)

**Model baked to finalize the internals**

model.bake()

**OUTPUT:**

Predicting Probabilities

model.probability([['A', 'B', 'C']])

0.11111111111111109

model.probability([['A', 'B', 'B']])

0.0

**12.AIM:Write a program to implement Hidden Markov Model.**

import numpy as np

import pandas as pd

import networkx as nx

import matplotlib.pyplot as plt

%matplotlib inline

# create state space and initial state probabilities

states = ['sleeping', 'eating', 'pooping']

pi = [0.35, 0.35, 0.3]

state\_space = pd.Series(pi, index=states, name='states')

print(state\_space)

print(state\_space.sum())

# create transition matrix

# equals transition probability matrix of changing states given a state

# matrix is size (M x M) where M is number of states

q\_df = pd.DataFrame(columns=states, index=states)

q\_df.loc[states[0]] = [0.4, 0.2, 0.4]

q\_df.loc[states[1]] = [0.45, 0.45, 0.1]

q\_df.loc[states[2]] = [0.45, 0.25, .3]

print(q\_df)

q = q\_df.values

print('\n', q, q.shape, '\n')

print(q\_df.sum(axis=1))

from pprint import pprint

# create a function that maps transition probability dataframe

# to markov edges and weights

def \_get\_markov\_edges(Q):

    edges = {}

    for col in Q.columns:

        for idx in Q.index:

            edges[(idx,col)] = Q.loc[idx,col]

    return edges

edges\_wts = \_get\_markov\_edges(q\_df)

pprint(edges\_wts)

**OUTPUT:**

sleeping 0.35

eating 0.35

pooping 0.30

Name: states, dtype: float64

1.0

sleeping eating pooping

sleeping 0.4 0.2 0.4

eating 0.45 0.45 0.1

pooping 0.45 0.25 0.3

[[0.4 0.2 0.4]

[0.45 0.45 0.1]

[0.45 0.25 0.3]] (3, 3)

sleeping 1.0

eating 1.0

pooping 1.0

dtype: float64

{('eating', 'eating'): 0.45,

('eating', 'pooping'): 0.1,

('eating', 'sleeping'): 0.45,

('pooping', 'eating'): 0.25,

('pooping', 'pooping'): 0.3,

('pooping', 'sleeping'): 0.45,

('sleeping', 'eating'): 0.2,

('sleeping', 'pooping'): 0.4,

('sleeping', 'sleeping'): 0.4}